Facultad de Matemáticas

Departamento de Ciencias de la Computación e Inteligencia Artificial

Trabajo Fin de Grado

Criptografía desde el punto de vista de la programación funcional

Daniel Rodríguez Chavarría

Entorno académico

El presente Trabajo Fin de Grado se ha realizado en el Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Sevilla. Ha sido supervisado por los profesores José A. Alonso Jiménez y María José Hidalgo Doblado. Se ha presentado el 6 de julio de 2016 ante un tribunal compuesto por profesores Joaquín Borrego, Félix Lara y Antonia M. Chávez.

Abstract

The present paper aims to explain cryptography from the point of view of functional programming. In order to accomplish this goal, we would use *Haskell* to write the code about many cryptosystems. In addition, we would verify the correctness of those functions by generating some random tests. We would deal with the following ciphers: *Cesar*, *Affine*, *Vigènere*, *Hill*, *Permutation*, *DES*, *RSA*, *ElGamal* and *Rabin*.

Índice general

Introducción

El hombre siempre ha intentado que la información confidencial permaneciera en secreto. Desde la Antigüedad, los militares, los espías, los diplomáticos e incluso los amantes clandestinos han utilizado claves y códigos cada vez más difíciles de descifrar provocando el nacimiento de una nueva ciencia: La criptografía. Etimológicamente, viene de las palabras griegas *kryptos*, que significa oculto; y *grafos*, que quiere decir escritura.

El arte de enmascarar, esconder y desvirtuar los mensajes para que la información no pueda llegar a los enemigos, o simplemente a ojos u oídos indiscretos, ha sido y es todo un ejercicio de ingenio. Desde el siglo V a.C., con la invención de la Escítala hasta los más avanzados protocolos de firmas digitales, la criptografía ha sido un tema de sumo interés para la humanidad. De hecho, ha tenido numerosas apariciones en la literatura de autores de la talla de *Julio Verne*, *Edgar Allan Poe* y *Arthur Conan Doyle*.

En la era de la información, una ciencia como la criptografía se ha ido convirtiendo en una herramienta esencial para el día a día. Desde encender la televisión hasta pagar con una tarjeta bancaria, pasando por navegar por Internet, los protocolos y sistemas criptográficos están presentes en el día a día de cada uno de nosotros. Dicha importancia es la motivación principal del desarrollo de este trabajo durante el cual podremos profundizar en el área de la criptografía.

Al encontrarnos en un entorno académico, hemos elegido el marco de la programación funcional para hacer dicho estudio. Ya que nos exige un grado de comprensión absoluto en todos los niveles de la programación. De esta forma, obtendremos a cambio diversas ventajas en la implementación como son la ausencia de efectos colaterales, procesos de depuración menos problemáticos y facilitar la verificación de las funciones definidas.

En ciencias de la computación, la programación funcional es un paradigma de programación declarativa basado en el uso de funciones matemáticas. La diferencia entre una función matemática y la noción de una "función" utilizada en la programación imperativa, es que las funciones imperativas pueden tener efectos secundarios. Por esta razón carecen de transparencia referencial; es decir, la misma expresión sintáctica puede resultar en valores diferentes en varios momentos de la ejecución del programa. Con código funcional, en cambio, el valor generado por una función depende exclusivamente de los argumentos de la función. Al eliminar los efectos secundarios se puede

entender y predecir el comportamiento de un programa mucho más fácilmente. El objetivo de este tipo de programación es conseguir lenguajes expresivos y matemáticamente elegantes, basados en la reescritura de expresiones.

A la hora de englobar este trabajo en un marco académico, podríamos hablar de él como un desarrollo paralelo al realizado en el proyecto *Cryptol*. Este proyecto trata de desarrollar un lenguaje en el que se facilite la verificación de propiedades y teoremas relacionados con la criptografía. Mediante el uso de este lenguaje se simplifican procesos como la generación aleatoria de vectores test o el estudio de equivalencia entre distintas implementaciones, procesos que veremos muy presentes durante todo el trabajo.

Nosotros nos serviremos del lenguaje puramente funcional *Haskell*, que se estudia en la Universidad de Sevilla en diversas asignaturas del Grado en Matemáticas. Este lenguaje está en constante crecimiento y se encuentra en la vanguardia de la programación actual. Es por esta razón que lo escogemos como sistema para la implementación de nuestras funciones.

Una de las grandes dificultades que nos hemos encontrado a la hora de implementar los distintos criptosistemas es el formato de los mensajes. Un mensaje, originalmente, está compuesto por caracteres que, a su vez, se encadenan formando palabras y frases. Pero trabajar con este tipo de dato no es, en absoluto, nada cómodo. Por esta razón asociamos los caracteres o incluso todo el mensaje a números enteros o binarios, con los cuales es mucho más sencillo trabajar (ver págs. 15 y 59). Es fácil ver la importancia que tiene el ser cuidadoso a la hora de definir las funciones que aplican (en ambas direcciones) las distintas asociaciones, y es por esto que trataremos de verificar la corrección de las mismas a medida que las implementamos.

Cabe destacar que nuestros procesos de verificación no son absolutos; es decir, no tienen una fiabilidad del cien por cien. Nosotros nos ayudaremos de la librería *Test.QuickCheck* para comprobar las distintas propiedades que definamos. Esta libreria comprobará que se verifica dicha propiedad para un conjunto de datos aleatorio generados por un generador que previamente tambiém habremos definido.

Partiendo de estas premisas el presente trabajo comienza hablando de uno de los códigos más famosos y simples que se conocen: el **cifrado César** (pág. 22). Sobre el que destacamos el estudio de criptoanálisis que hemos podido hacer; que, a parte del desencriptado por fuerza bruta, se ofrece a realizar un ataque basado en el análisis de frecuencias.

A continuación, trataremos el **cifrado afín** (pág. 27), que generaliza al César. Este criptosistema, también basado en la aritmética modular, introduce la necesidad de calcular el inverso de un número dado para un cierto módulo. Esta función será recurrente durante el desarrollo del trabajo ya que la aritmética modular toma mucha importancia en la gran mayoría de los criptosistemas.

Como combinación de diversos cifrados César surge el **cifrado de Vigenère** (pág. 32). Para comprobar la corrección del mismo, nos hemos ayudado de los generadores

definidos en la página 23, que ya nos sirvieron para el cifrado César y el afín.

El siguiente criptosistema que hemos estudiado es el **cifrado de Hill** (pág. 35), que, al usar matrices como claves, presenta una nueva dificultad: la generación aleatoria de matrices. Para más inri, dichas matrices deben ser invertibles, y es necesario calcular la matriz inversa para el descifrado. Este cálculo lo hemos hecho mediante la matriz adjunta traspuesta multiplicada por el inverso (modular) del determinante (que lo hemos calculado por *Laplace*).

Como generalización del cifrado que hacía la escítala surge el **cifrado por Permutaciones** (pág. 40). Cabe destacar que, debido a su simplicidad es el único para el que no resulta necesaria una asociación entre mensaje y números. Esto se debe a que trabaja directamente sobre los caracteres, intercambiándolos entre sí.

El último ejemplo de criptosistema de clave privada que veremos es el **cifrado DES** (pág. 43). Éste cimenta la base de los criptosistemas simétricos más usados en la actualidad como el 3DES o el AES. Para la implementación del mismo nos ha hecho falta definir dos tipos de asociaciones entre mensajes y conjuntos de números binarios, así como múltiples funciones auxiliares. Debido a la complejidad del algoritmo se ha ido implementando a la misma vez que lo describíamos.

En lo que se refiere a la criptografía de clave pública hemos tratado con tres de los ejemplos más famosos basados en tres problemas distintos: La factorización, el problema del logaritmo discreto (DLP) y el problema de las raíces cuadradas (pág. 58). Estos protocolos son algo más complejos que los anteriores y se vio la necesidad de incluir en la implementación el uso de mónadas.

El primero de los cifrados asimétricos que surgió se basaba en el problema de factorizar el producto de dos números primos muy grandes. Se conoce como **criptosistema RSA** (pág. 62) y es uno de los más usados en la actualidad. A la hora de implementarlo hemos encontrado un gran problema: la generación aleatoria de números primos. Aunque con la ayuda de la exahustiva librería Codec.Crypto.RSA.Exceptions pudimos suplir este inconveniente.

En segundo lugar, tratamos el **criptosistema de ElGamal** (pág. 68) basado en el problema del logaritmo discreto (DLP). Durante la implementación encontramos una dificultad a la hora de escoger de un número no primo su mayor factor. Para ello aprovechamos dos implementaciones de la librería factory, sobre las cuales hicimos un estudio de eficiencia y escogimos en consecuencia la más rápida.

Además, en los dos últimos criptosistemas vimos la importancia y peculiaridad de la aritmética modular, que nos permitió definir un tipo de exponenciación mucho más eficiente que la usual. Para más información ver la página 65.

Para terminar, el problema de las raíces cuadradas en aritmética modular da origen al **criprosistema de Rabin** (pág. 75). Del cual me gustaría destacar que la desencriptación no es única, tenemos cuatro posibilidades. Esto se debe a que, generalmente, en aritmética modular existen cuatro posibles raíces cuadradas de un número dado.

De esta forma queda completado nuestro repaso a la criptografía. Destacando que objetivo del trabajo es facilitar la comprensión de los distintos criptosistemas así como el proceso de verificación de las funciones que los implementan. Quedándose abierta como vía de investigación la implementación de otros criptosistemas ya descubiertos o de nuestra propia invención, que como estipularon los principios de Kerckhoffs deben cumplir que:

- Si el sistema no es teóricamente irrompible, al menos debe serlo en la práctica.
- La efectividad del sistema no debe depender de que su diseño permanezca en secreto.
- La clave debe ser fácilmente memorizable de manera que no haya que recurrir a notas escritas.
- Los criptogramas deberán dar resultados alfanuméricos.
- El sistema debe ser operable por una única persona.
- El sistema debe ser fácil de utilizar.

Aunque algunos puedan parecer obsoletos, son una buena cimentación para desarrollar nuestro propio sistema criptográfico que quizás revolucione el mundo. Para lo cual no hay que darse nunca por vencido, pues como dijo *Steve Jobs*:

> *"A veces, cuando se innova, se cometen errores. Es mejor admitirlos rápidamente, y seguir adelante con la mejora de tus otras innovaciones."*

Capítulo 1

Criptografía de clave privada

1.1. Descripción

La **criptografía de clave privada**, también conocida como **criptografía simétrica**, se basa en un método o algoritmo en el cual se usa una misma clave para el cifrado y descifrado del mensaje.

Como sabemos, el mensaje se transmite del emisor al receptor, y son éstos quienes se servirán de la clave para cifrar y, posteriormente, descifrar el mensaje. Por tanto, las dos partes han de llegar al un acuerdo con anterioridad sobre la clave que van a usar para comunicarse. Este detalle de la criptografía de clave privada es uno de los más delicados y lo trataremos unas líneas más abajo.

Según el principio de *Kerkhoff*, la seguridad del sistema debe recaer en la seguridad de la clave, entendiendo como públicose el resto de datos (incluso el propio algoritmo de encriptación). En este capítulo veremos distintos ejemplos, como el código *César*, el *Hill* o el *DES*. Todos ellos procuran, en mayor o menor medida, cumplir con ésta propiedad.

En la década de los cuarenta, *Claude Shanon* demostró que, para tener una seguridad completa, los sistemas de clave privada debían usar claves que tuviesen, como mínimo, la misma longitud que el mensaje a encriptar. Teniendo esto en cuenta y suponiendo que el canal entre el emisor y el receptor no es seguro, ¿cómo hacen para intercambiar la clave? La primera opción que nos puede acudir a la cabeza es encriptar también la clave, pero es fácil ver que este camino nos lleva a un callejón sin salida pues nos volvemos a encontrar con el mismo problema: ¿Cómo intercambiar la segunda clave que hemos usado para cifrar la primera clave? En realidad éste es un problema sin solución dentro de la criptografía simétrica y se convierte en la mayor desventaja de cualquier criptosistema de este tipo.

Suponiendo que el intercambio de claves se ha realizado por un canal seguro, es fácil darse unta que toda la seguridad del criptosistema recae en la propia clave privada. Así que el sistema será más seguro en tanto el abanico de claves posibles sea más grande; es decir, el espacio de claves sea mayor.

En la actualidad, los ordenadores pueden descifrar claves con extrema rapidez, dejando obsoletos a muchos sistemas de cifrado. Por ejemplo, el algoritmo de cifrado *DES* usa una clave de 56 bits, lo que significa que es espacio de claves es del orden de 2^{56} (más de 70 mil billones de posibilidades). A priori, este número puede parecer grande pero un ordenador podría comprobar todo el conjunto de claves en cuestión de días.

A raíz de la introducción de la informática en la criptografía, los sistemas de clave privada han ido evolucionando para que este espacio de claves sea mucho mayor. Por ejemplo, el *3DES* usa claves de 128 bits; es decir, existen 2^{128} claves posibles (un número ya impracticable incluso para los ordenadores).

En este capítulo implementaremos en *Haskell* algunos de los criptosistemas más conocidos de la criptografía de clave privada. Además, haremos hincapié en la verificación de que las funciones sean correctas; así como en la generación aleatoria de mensajes y claves. Para facilitar la comprensión de los criptosistemas vamos a fijar una misma notación para todos:

- El mensaje que quiere enviar el emisor al receptor: m
- La clave que sirve para encriptar y desencriptar: k
- La función encriptadora, que toma dos valores (clave y mensaje): $F(\cdot, \cdot)$
- El mensaje encriptado: $F(k, m) = m'$
- Una función desencriptadora: $G(\cdot, \cdot)$, que es inversa de la anterior cumpliendo que $G(k, m') = m$

1.2. Primeros pasos

Los primeros pasos que tenemos que aclarar es qué tipos de mensajes vamos a encriptar; es decir, qué caracteres van a estar permitidos y cuáles no. Por simplicidad, trabajaremos con mensajes que están escritos únicamente en mayúsculas y que no tienen espacios. Por ejemplo, la cadena "UNMENSAJEVALIDO" es un mensaje válido y "Noes unmenSaJe" no lo es

Huelga decir que las definiciones son análogas si queremos admitir los espacios y las minúsculas. Sin embargo, por convenio, adoptaremos esta metodología. Para ver si un mensaje es válido en Haskell necesitamos definir la lista `abecedario` que contiene a las letras mayúsculas de la A a la Z:

```
abecedario :: String
abecedario = ['A'..'Z']
```

Y la función que comprueba si un mensaje no es vacío y sólo tiene mayúsculas es:

```
enAbecedario :: String -> Bool
enAbecedario xs = not (null xs) && all (\x -> elem x abecedario) xs
```

Por ejemplo,

```
ghci> enAbecedario "HOLA"
True
ghci> enAbecedario "hola"
False
ghci> enAbecedario "AÑOS"
False
```

En criptografía, para poder trabajar con los mensajes y aplicarles ciertas funciones, es más sencillo asociar a cada letra un número y trabajar con él. Para ello descomponemos cada mensaje en las letras que lo componen. Trabajamos con cada letra de forma independiente (o en grupos) y volvemos a "juntar" todas las letras transformadas. De esta forma obtendremos el nuevo mensaje encriptado.

Principalmente podemos distinguir dos formas de asociar las letras a números: La primera opción es que cada letra corresponde a un entero en módulo 26 dado por su posición en el alfabeto. La segunda, hace corresponder a cada letra un número binario, determinado también por su posición en el alfabeto.

Primera asociación

A la primera letra del alfabeto, la A, le asociamos el número 0; a la B, el 1; a la C, el 2; y así sucesivamente hasta llegar a la Z, que le corresponde el número 25.

Desde el punto de vista computacional necesitamos implementar funciones que hagan esta asociación. Así que la función que a cada número n, entre el 0 y el 25, le hace corresponder la n–ésima letra mayúscula es:

```
int2char :: Int -> Char
int2char n = chr (n + 65)
```

Por ejemplo:

```
ghci> int2char  0
'A'
ghci> int2char  1
'B'
ghci> int2char 25
'Z'
```

Mientras que la función que a cada letra le hace corresponder la posición que ocupa en el alfabeto es:

```
char2int :: Char -> Int
char2int c = ord c - 65
```

Unos ejemplos de cómo actúa son:

```
ghci> char2int 'A'
0
ghci>  char2int 'B'
1
ghci> char2int 'Z'
25
```

El siguiente paso es definir estas mismas funciones pero que actúen sobre el mensaje entero. En definitiva, la función que a cada lista de posiciones le asocia el mensaje correspondiente es:

```
int2str :: [Int] -> String
int2str = map int2char
```

Y la función que, dado un mensaje, devuelve la lista de las posiciones de cada letra es:

```
str2int :: String -> [Int]
str2int = map char2int
```

Que actúan de la siguiente forma:

```
ghci> int2str [0,2,1,25]
"ACBZ"
ghci> str2int "ACBZ"
[0,2,1,25]
```

Segunda asociación

Anteriormente sólo nos estaba permitido escribir mensajes usando el abecedario en mayúsculas. En cambio, esta segunda asociación es algo más general que la anterior, por lo que no limitaremos los caracteres que podemos usar en el mensaje o en la clave. A la primera letra del alfabeto, la A, le asociamos el número binario $[1,0,0,0,0,0,1,0]$; a la B, el $[0,1,0,0,0,0,1,0]$ y así sucesivamente, llegando a asociar todos los caracteres con un número binario. Vamos a describir con exactitud dicha asociación comentando el código que la ejecuta.

En primer lugar, asociaremos números enteros a sus códigos binarios correspondientes; ya que, gracias a las funciones `ord` y `chr` podemos asociar fácilmente caracteres a enteros. De esta forma, el código de la función que dado un entero nos devuelve su código binario es:

```
int2bin :: Int -> [Int]
int2bin n | n < 2     = [n]
          | otherwise = n `mod` 2 : int2bin (n `div` 2)
```

Como veremos en los ejemplos, el coeficiente de la potencia cero del dos va en la primera posición (izquierda), mientras que el de la mayor va en el lugar más a la derecha:

```
ghci> int2bin 2
[0,1]
ghci> int2bin 24
[0,0,0,1,1]
```

Sin embargo, para la función inversa existen distintas variantes, por recursión, comprensión y plegados:

```
bin2intR :: [Int] -> Int
bin2intR [] = 0
bin2intR (x:xs) = x + 2 * bin2intR xs

bin2intC :: [Int] -> Int
bin2intC xs = sum [x*2^n | (x,n) <- zip xs [0..]]

bin2int :: [Int] -> Int
bin2int =  foldr (\x y -> x + 2*y) 0
```

Nosotros nos quedaremos con esta última, que es la más eficiente. Por ejemplo:

```
ghci> bin2intR [0,1]
2
ghci> bin2intR [0,1,0,1,0,0,1,1,0,1]
714
```

El siguiente paso es comprobar que esta asociación es correcta. Para ello, implementaremos la siguiente propiedad: Al pasar un número natural a binario con `int2bin` y el resultado a decimal con `bin2int` se obtiene el número inicial. Cuyo código es:

```
prop_int2bin :: Int -> Bool
prop_int2bin x =
   bin2int (int2bin y) == y
   where y = abs x
```

Que al ejecutar en consola:

```
ghci> quickCheck prop_int2bin
+++ OK, passed 100 tests.
```

Así que podemos aumentar nuestro grado de confianza en las funciones que acabamos de implementar. Finalmente, debemos terminar de asociar un mensaje con un número binario. Para ello, nos fijaremos en cada carácter por separado, dando el binario que le corresponde y concatenándolos todos. A este proceso le llamaremos codificación.

Antes de nada, hace falta definir una función auxiliar que nos ayudará en la codificación. Dicha función hará que todos los números binarios sean de longitud 8. Este detalle se debe a que no hay más de 255 caracteres.

```
creaOcteto :: [Int] -> [Int]
creaOcteto bs =  take 8 (bs ++ repeat 0)
```

De esta manera, la implementación de la función que codifica un mensaje en un binario es:

```
codifica :: String -> [Int]
codifica =  concatMap (creaOcteto . int2bin . ord)
```

Para realizar el proceso inverso necesitaremos una función auxiliar que nos separe nuestro número binario (seguramente muy grande) en distintas listas de longitud ocho, que corresponderán a los números binarios originales (antes de concatenarlos):

```
separaOctetos :: [Int] -> [[Int]]
separaOctetos [] = []
separaOctetos bs =
    take 8 bs : separaOctetos (drop 8 bs)
```

Por ejemplo:

```
ghci> separaOctetos [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0]
[[1,0,0,0,0,1,1,0],[0,1,0,0,0,1,1,0]]
```

Apoyándonos en esta función, el código de la función que descodifica un número binario en el mensaje original es:

```
descodifica :: [Int] -> String
descodifica =  map (chr . bin2int) . separaOctetos
```

Tercera asociación

Debido al proceso de encriptación del DES nos surge la necesidad de introducir otra asociación entre enteros y números binarios. El objetivo es traducir cada número decimal a uno binario de, como mínimo, longitud cuatro. Así como realizar el proceso inverso. Comenzaremos definiendo una función que, dado un entero, nos da el binario que le corresponde (sin tener en cuenta la longitud de salida):

```
int2bin4' :: Integral a => a -> [a]
int2bin4' n | n < 2     = [n]
            | otherwise = int2bin4' (n `div` 2) ++ [n `mod` 2]
```

Por ejemplo:

```
ghci> int2bin4' 2
[1,0]
ghci> int2bin4' 5
[1,0,1]
```

Para conseguir que la longitud de salida sea como mínimo de cuatro, implementamos la siguiente función:

```
largo4 :: Num a => [a] -> [a]
largo4 xs | length xs >= 4 = xs
          | otherwise      = largo4 (0:xs)
```

Lo único que hace esta función es añadir ceros al principio de una lista si ésta es de longitud menor que cuatro. Veamos a continuación unos ejemplos:

```
ghci> largo4 [1,0]
[0,0,1,0]
ghci> largo4 [1,1,0,1]
[1,1,0,1]
```

Ayudándonos de las dos funciones anteriores ya es simple dar el código de una que nos transforme los números enteros positivos en binario de longitud mayor o igual que cuatro:

```
int2bin4 :: Integral a => a -> [a]
int2bin4 = largo4 . int2bin4'
```

Mientras que la función inversa viene dada por:

```
bin2int4 :: [Int] -> Int
bin2int4 xs = foldr (\x y -> x + 2*y) 0 (reverse xs)
```

Algunos ejemplos donde las hemos usado son:

```
ghci> int2bin4 3
[0,0,1,1]
ghci> bin2int4 it
3
ghci> bin2int4 [1,0,0,1,1]
19
ghci> int2bin4 it
[1,0,0,1,1]
```

Para ver que la asociación es correcta, vamos a definir la propiedad de que al pasar un número natural a binario con `int2bin4` y el resultado a decimal con `bin2int4` obtenemos el número inicial:

```
prop_int2bin4 :: Int -> Bool
prop_int2bin4 x =
   bin2int4 (int2bin4 y) == y
   where y = abs x
```

Y la comprobación en la consola queda:

```
ghci> quickCheck prop_int2bin4
+++ OK, passed 100 tests.
```

También nos hará falta una función que codifique los booleanos, emparejando el *Verdadero* con el 1 y el *Falso* con el 0. Esta función es:

```
bool2int :: Bool -> Int
bool2int True  = 1
bool2int False = 0
```

1.3. Criptosistema César

1.3.1. Introducción

El **cifrado César** recibe su nombre en honor a *Julio César* quien, según *Suetonio*, usaba este sistema para enviar mensajes secretos durante sus campañas militares sin que lo descubrieran.

Lo que *Julio César* pensó fue en sustituir cada letra del abecedario por la letra situada tres lugares más allá en el orden alfabético. De esta forma, la letra A se convertiría en la D, la B en la E y así sucesivamente.

Para nosotros, el cifrado César, también conocido como **cifrado por desplazamiento**, **código de César** o **desplazamiento de César**, es una de las técnicas de cifrado más simples y más usadas. Es un tipo de cifrado por sustitución en el que una letra en el texto original es reemplazada por otra letra que se encuentra un número fijo de posiciones más adelante en el alfabeto.

1.3.2. Descripción matemática

Dados un mensaje $m = m_1 m_2 \ldots$, donde cada m_i es el entero correspondiente a la i–ésima letra del mensaje, y la clave privada $k \in \mathbb{N}$, el procedimiento de cifrado que debemos de llevar a cabo consiste en sumar ambos números módulo 26. De esta forma la función encriptadora F se define:

$$F(m_i, k) = m_i + k \mod 26$$

Mientras que el descifrado que realiza el receptor del mensaje encriptado m' no es más que restar la clave k al entero que le corresponde a cada letra de m'. Quedando la función desencriptadora G definida de la siguiente forma:

$$G(m'_i, k) = m'_i - k \mod 26$$

1.3.3. Implementación

El código de la función encriptadora F que recibe como argumentos la clave k (un entero) y el mensaje m (una lista de letras) es:

```
cesarF :: Int -> String -> String
cesarF k m = int2str [(n+k) `mod` 26 | n <- str2int m]
```

De la misma forma, el código de la función G queda:

```
cesarG :: Int -> String -> String
cesarG k = cesarF (-k)
```

Unos ejemplos de cómo funcionan estas funciones son:

```
ghci> cesarF 3 "TODOPARANADA"
"WRGRSDUDQDGD"

ghci> cesarG 3 "WRGRSDUDQDGD"
"TODOPARANADA"
```

1.3.4. Corrección

Un criptosistema es **correcto** si al encriptar un mensaje y después desencriptarlo obtenemos el original.

Usaremos QuickCheck para comprobar la corrección del criptosistema. Lo que hace QuickCheck es verificar que nuestras funciones al aplicarse sucesivamente nos dan el mensaje original para algunos ejemplos; no confundir con una demostración.

Nos va a hacer falta definir un generador de mensajes (que sólo use las letras A,B,...,Z). Para ello definimos un generador aleatorio de letras que pertenezcan a la lista `abecedario` que ya definimos en la página 15:

```
genLetra :: Gen Char
genLetra = elements abecedario
```

Con `genLetra` se generan las siguientes letras:

```
ghci> generate genLetra
'K'
ghci> generate genLetra
'Y'
```

El siguiente paso es definir un generador de mensajes apoyádonos del generador aleatorio de letras y ver cómo funciona:

```
mensaje :: Gen String
mensaje = listOf genLetra
```

Por ejemplo,

```
ghci>  generate mensaje
"WAIPNGKPOZDETCBVGXNUSSMQJAKG"
```

Finalmente, ya estamos en disposición de definir la propiedad que queremos comprobar:

```
prop_CorreccionDeCesar :: Int -> Property
prop_CorreccionDeCesar k =
    forAll mensaje (\m -> m == cesarG k (cesarF k m))
```

Que al comprobar con QuickCheck nos queda:

```
ghci> quickCheck prop_CorreccionDeCesar
+++ OK, passed 100 tests.
```

Así que podemos decir que nuestra implementación seguramente sea correcta.

1.3.5. Criptoanálisis

El criptosistema César en la práctica no es muy usado porque es muy débil. Sólo existen 26 claves posibles en nuestro caso, y en general, hay tantas claves como letras tenga el abecedario que usemos. Por tanto, es normal que la primera forma que se nos ocurra de atacarlo o romperlo es por fuerza bruta; es decir, probando todas las claves. La función que genera todos los posibles mensajes m'correspondiente al mensaje m es

```
desencriptaCesarBruta :: String -> [String]
desencriptaCesarBruta m = [cesarG k m | k <- [0..25]]
```

Pero, evidentemente, esto es muy ineficaz. Así que vamos a tratar de afinar un poco el ataque. Para ello nos basaremos en la frecuencia o repetición de cada letra del mensaje encriptado. Así que necesitamos de una función que nos calcule cuántas apariciones ha habido de cada letra. Empezamos definiendo la función `aparicion` tal que `(aparicion a xs)` es el número de veces se repite el carácter `a` en la cadena `xs`:

```
aparicion :: Char -> String -> Int
aparicion a xs = length (filter (==a) xs)
```

Y nos basta calcular las apariciones de todas las letras del abecedario en nuestro mensaje:

```
frecuencias :: String -> [(Int,Char)]
frecuencias m = [(aparicion a m,a) | a <- abecedario]
```

Unos ejemplos de cómo funcionan son:

```
ghci> aparicion 'A' "AMAZONAS"
3
ghci> frecuencias "TODOPARANADA"
[(4,'A'),(0,'B'),(0,'C'),(2,'D'),(0,'E'),(0,'F'),(0,'G'),(0,'H'),
 (0,'I'),(0,'J'),(0,'K'),(0,'L'),(0,'M'),(1,'N'),(2,'O'),(1,'P'),
 (0,'Q'),(1,'R'),(0,'S'),(1,'T'),(0,'U'),(0,'V'),(0,'W'),(0,'X'),
 (0,'Y'),(0,'Z')]
```

Una vez que sabemos las frecuencias de cada letra usamos el dato de que en el castellano y en el Inglés la letra más usada es la *E*. Por tanto, tiene sentido pensar que (si el texto es largo) la letra más repetida de nuestro mensaje encriptado corresponderá a la *E* en el mensaje original. Basta desencriptar el mensaje con esta premisa y tenemos un buen candidato a mensaje desencriptado. La función que selecciona las frecuencias más altas es:

```
frecMayores :: String -> String
frecMayores m = [y | (x,y) <- frecuencias m, x == frecMayor]
    where frecMayor = maximum [aparicion a m | a <- abecedario]
```

La implementación de este desencriptador es como sigue:

```
desencriptaCesarFino :: String -> [String]
desencriptaCesarFino m =
   [cesarG ((char2int k)-4) m | k <- frecMayores m]
```

Un ejemplo de cómo actuarían estos desencriptadores es:

```
ghci> desencriptaCesarBruta "QTGZJSTXNGWJAJITXAJHJXGZJST"
["QTGZJSTXNGWJAJITXAJHJXGZJST","PSFYIRSWMFVIZIHSWZIGIWFYIRS",
"OREXHQRVLEUHYHGRVYHFHVEXHQR","NQDWGPQUKDTGXGFQUXGEGUDWGPQ",
"MPCVFOPTJCSFWFEPTWFDFTCVFOP","LOBUENOSIBREVEDOSVECESBUENO",
"KNATDMNRHAQDUDCNRUDBDRATDMN","JMZSCLMQGZPCTCBMQTCACQZSCLM",
"ILYRBKLPFYOBSBALPSBZBPYRBKL","HKXQAJKOEXNARAZKORAYAOXQAJK",
"GJWPZIJNDWMZQZYJNQZXZNWPZIJ","FIVOYHIMCVLYPYXIMPYWYMVOYHI",
"EHUNXGHLBUKXOXWHLOXVXLUNXGH","DGTMWFGKATJWNWVGKNWUWKTMWFG",
"CFSLVEFJZSIVMVUFJMVTVJSLVEF","BERKUDEIYRHULUTEILUSUIRKUDE",
```

```
"ADQJTCDHXQGTKTSDHKTRTHQJTCD","ZCPISBCGWPFSJSRCGJSQSGPISBC",
"YBOHRABFVOERIRQBFIRPRFOHRAB","XANGQZAEUNDQHQPAEHQOQENGQZA",
"WZMFPYZDTMCPGPOZDGPNPDMFPYZ","VYLEOXYCSLBOFONYCFOMOCLEOXY",
"UXKDNWXBRKANENMXBENLNBKDNWX","TWJCMVWAQJZMDMLWADMKMAJCMVW",
"SVIBLUVZPIYLCLKVZCLJLZIBLUV","RUHAKTUYOHXKBKJUYBKIKYHAKTU"]

ghci> desencriptaCesarFino "QTGZJSTXNGWJAJITXAJHJXGZJST"
["LOBUENOSIBREVEDOSVECESBUENO"]
```

1.4. Criptosistema afín

1.4.1. Introducción

El **cifrado afín** (o **cifrado de transformación afín** o **cifrado monoalfabético genérico**) es un tipo de cifrado por sustitución mono–alfabético en el que a cada carácter del mensaje se le aplica una función matemática afín $(ax + b)$ en aritmética modular. Que un cifrado sea **mono–alfabético** quiere decir que a cada letra le corresponde una única letra encriptada, siendo esta correspondecia biunívoca. En realidad, el cifrado afín no es más que una extensión del cifrado César visto anteriormente y, por esto, se considera como parte de la criptografía clásica. En la actualidad, no tiene ninguna aplicación porque es un criptosistema bastante débil.

1.4.2. Descripción matemática

Dados un mensaje $m = m_1 m_2 \ldots$, donde cada m_i es el entero correspondiente a la i–ésima letra del mensaje, y una clave constituida por un par de números enteros $k = (k_1, k_2)$ el procedimiento de cifrado consiste en multiplicar cada entero m_i por k_1 y después sumarle k_2, todo en módulo 26. Por tanto la función encriptadora F se define por

$$F(m_i, k) = (k_1 \times m_i) + k_2 \quad \text{mód } 26$$

Esta definición de la función encriptadora nos da una restricción sobre la clave que podemos usar: $k_1 \neq 0$. Pues, de no ser así, la función F sería la función constante k_2.

Para que el receptor, que conoce la clave $k = (k_1, k_2)$ con anterioridad, pueda desencriptar el mensaje encriptado m' hay que realizar previamente un cálculo en aritmética modular, que consiste en hallar el inverso de k_1. Este inverso lo denotaremos por k_1^{-1}. Esto nos plantea un problema algebraico muy importante: ¿nos sirve cualquier $k_2 \in \mathbb{F}_{26}$? La respuesta es que no. Esto se debe a que, en el anillo finito $\mathbb{F}_{26}$, no todos los elementos son unidades. Por tanto, hemos encontrado una segunda restricción de las claves que podemos utilizar: k_1 tiene que ser primo con 26. Luego el descifrado de m' consiste en:

$$G(m_i', k) = (m_i' - k_2) \times k_1^{-1} \quad \text{mód } 26$$

1.4.3. Clasificación

En función de la clave $k = (k_1, k_2)$ que usemos los cifrados afines se pueden clasificar en tres grupos:

- Si $k_1 = 1$, entonces tendremos un criptosistema ya conocido: el cifrado César con clave k_2.

- Si $k_2 = 0$, que entonces se dirá que es un **cifrado por decimación pura**.
- Si $k_1 \neq 1$ y $k_2 \neq 0$, que entonces tendremos un verdadero cifrado por sustitución afín.

1.4.4. Implementación

El código de la función encriptadora F que recibe como argumentos la clave (k_1, k_2) (un par de enteros) y el mensaje m (una lista de letras) es:

```
afinF :: (Int,Int) -> String -> String
afinF (k1,k2) m = int2str [(n*k1+k2) `mod` 26 | n <- str2int m]
```

Como explicamos anteriormente, para desencriptar necesitamos calcular un inverso módulo 26. Así que vamos a implementar la función `invmod` que recibirá un entero a y el módulo n en el que trabajaremos y nos devolverá su inverso modular sólo si a y n son primos entre sí. Nos apoyaremos del algoritmo extendido de *Euclides* por lo tanto también debemos de implementarlo. La función `mcdExt` recibirá dos números enteros (x,y) y nos dirá quiénes son (a,b,g) tales que g es el máximo común divisor de ex e y y $x * a + y * b = g$:

```
mcdExt :: Integral a => a -> a -> (a,a,a)
mcdExt a 0 = (1, 0, a)
mcdExt a b = (t, s - q * t, g)
    where (q, r)    = a `quotRem` b
          (s, t, g) = mcdExt b r
```

Y el inverso modular se define:

```
invMod :: Integer -> Integer -> Integer
invMod a m | x < 0     = x + m
           | otherwise = x
    where (x, _, _) = mcdExt a m
```

Entonces, el código de la función G desencriptadora nos queda de la siguiente forma:

```
afinG :: (Int,Int) -> String -> String
afinG (k1,k2) m' =
    int2str [((n-k2) * invMod k1 26) `mod` 26 | n <- str2int m']
```

Unos ejemplos de cómo se ejecutan estas funciones son:

```
ghci> afinF (1,0) "LAGEOMETRIAESELARTEDEPENSARBIENYDIBUJARMAL"
"LAGEOMETRIAESELARTEDEPENSARBIENYDIBUJARMAL"
ghci> afinF (5,2) "LAGEOMETRIAESELARTEDEPENSARBIENYDIBUJARMAL"
"FCGWUKWTJQCWOWFCJTWRWZWPOCJHQWPSRQHYVCJKCF"
ghci> afinG (5,2) "FCGWUKWTJQCWOWFCJTWRWZWPOCJHQWPSRQHYVCJKCF"
"LAGEOMETRIAESELARTEDEPENSARBIENYDIBUJARMAL"
```

1.4.5. Corrección

Nos vuelve a hacer falta definir un generador de mensajes idéntico al anterior. Además haremos que genere también las claves, cumpliendo que la primera componente del par sea primo con 26. Empezaremos implementando una función que compruebe si un número es primo con 26:

```
primocon26 :: Int -> Bool
primocon26 n = gcd n 26 == 1
```

Usando el generador aleatorio de mensajes definido en la página 23 podemos implementar un generador de mensajes y claves correctas, es decir, que la clave tenga su primer número primo con 26:

```
mensajeYclaveA :: Gen (String,(Int,Int))
mensajeYclaveA = do m <- mensaje
                    k1 <- suchThat (choose (0,26)) primocon26
                    k2 <- choose (0,26)
                    return (m,(k1,k2))
```

Y la propiedad que comprueba que el mensaje encriptado y posteriormente desencriptado es el original la implementamos así:

```
prop_CorreccionAfin :: Int -> Property
prop_CorreccionAfin k =
    forAll mensajeYclaveA (\ (m,k) -> m == afinG k (afinF k m))
```

Que al ejecutar el QuickCheck:

```
ghci> quickCheck prop_CorreccionAfin
+++ OK, passed 100 tests.
```

1.4.6. Criptoanálisis

El criptosistema afín es otro ejemplo de criptosistema bastante débil. Esto se debe a su reducido número de claves posibles; por ejemplo, con un alfabeto de 26 letras el número máximo de claves distintas que podemos crear es $12 * 26 = 312$. Un número bastante bajo teniendo en cuenta que trabajamos con ordenadores. Por tanto una manera de atacarlo bastante factible sería probar con la fuerza bruta; es decir, probar con las 312 claves. Nosotros vamos a intentar ir un poco más allá y ser más sutiles. Nos aprovecharemos de que este cifrado es atacable mediante el análisis de frecuencias.

El primer paso será encontrar las dos letras más repetidas de nuestro mensaje encriptado, y para ello definiremos la siguiente función:

```
letrasFrecuentes :: String -> String
letrasFrecuentes m = take 2 (ordena2 (frecuencias m))
```

Donde la función `frecuencias` está definido en la página 24 y `ordena2` simplemente ordena las frecuencias de mayor a menor. Su definición es

```
ordena2 :: [(Int,Char)] -> String
ordena2 xs = map snd (sortBy (flip compare) xs)
```

Como sabemos, en el castellano, las dos letras más usadas son la E y después la A. Así que, tiene sentido pensar que las letras que aparecen más veces en el mensaje encriptado corresponden en realidad a la E y la A. De esta forma obtenemos el siguiente sistema; donde x e y son la primera y la segunda letras más repetidas en el texto cifrado, respectivamente, y k_1 y k_2, las claves que queremos averiguar:

$$\begin{cases} 4k_1 + k_2 = x \\ 0k_1 + k_2 = y \end{cases}$$

Es fácil ver del sistema que $k_2 = y$ así que sólo bastaría despejar k_1. Para ello implementamos la siguiente función:

```
despejak1 :: Int -> Int -> Int
despejak1 x y =
    invMod (head [k1 | k1 <- [1..25],
                       k1 * (x - y) `mod` 26 == 4,
                       gcd k1 26 == 1])
           26
```

Juntando ambas funciones obtenemos nuestro candidato a clave:

```
posibleClave :: String -> (Int,Int)
posibleClave m = (despejak1 x y,y)
    where [x,y] = str2int (letrasFrecuentes m)
```

Ahora basta con descifrar el mensaje secreto usando esta clave:

```
desencriptaAfin :: String -> String
desencriptaAfin m = afinG (posibleClave m) m
```

Algo que debemos tener en cuenta cuando usemos el análisis de frecuencias es que si el texto encriptado es sólo una palabra o una frase corta seguramente las frecuencias no sean las esperadas. Y entonces nuestro desencriptador no funcionará. Además, debemos de saber el idioma en el que está escrito. En nuestro caso hemos implementado un descifrador para mensajes escritos originalmente en castellano. Para acabar veamos nuestro desencriptador en acción:

```
ghci> desencriptaAfin "FCGWUKWTJQCWOWFCJTWRWZWPOCJHQWPSRQHYVCJKCF"
"LAGEOMETRIAESELARTEDEPENSARBIENYDIBUJARMAL"
```

1.5. Criptosistema Vigenère

1.5.1. Introducción

El **cifrado de Vigenère** es un tipo de cifrado por sustitución poli–alfabético basado en el cifrado Césa. Un **cifrado poli–alfabético** quiere decir que a una letra del alfabeto original le pueden corresponder varias del mensaje encriptado. Por ejemplo, la primera vez que aparece la A se sustituye por la B, pero quizás la próxima vez que aparezca la A sea sustituida por una Z o una F.

Como pasa tantas veces en la historia, lo que actualmente conocemos como cifrado de Vigenère fue descrito en realidad por Giovan Battista Belasso en su libro de 1553 *La cifra del Sig. Giovan Battista Bel[l]aso, gentil'huomo bresciano, nuovamente da lui medesimo ridotta à grandissima brevità et perfettione*. Sin embargo, en el siglo XIX fue incorrectamente atribuido a Blaise de Vigenère, y por ello aún se le conoce como el "cifrado Vigenèr".

Este criptosistema fue considerado durante varios siglos como seguro e infranqueable lo que le hizo valedor del apodo de código indescifrable. Tendríamos que esperar hasta el siglo XIX para que el método Kasiski lo resolviera.

El cifrado Vigenère usa un método de sustitución equivalente al del código César, un desplazamiento de cada caractér en módulo 26. Pero la diferencia (y lo que lo hace mucho más robusto) es que dicho desplazamiento viene indicado por una clave que se escribe cíclicamente bajo el mensaje.

1.5.2. Descripción matemática

Dado un mensaje $m = m_1 m_2 ...$ y una clave $k = k_1 k_2 ...$ donde cada m_i y k_i es el entero correspondiente a la letra i–ésima del mensaje o clave, respectivamente. El procedimiento de cifrado consiste en repetir cíclicamente la clave, colocar esta cadena debajo de nuestro mensaje y sumar cada letra del mensaje con la que le corresponda de la clave.

Por ejemplo, suponiendo que la clave sea de longitud 2, consiste en aplicar el código César de clave k_1 a las letras impares y el código César de clave k_2 a las letras pares. Quedando de la siguiente manera si ON es la clave y ENCENDIDO el mensaje que queremos encriptar:

$$\begin{aligned} F_{\text{Cesar}}\,(\text{O}, \text{ECNIO}) &= \text{SQBWC} \\ F_{\text{Cesar}}\,(\text{N}, \text{NEDD}) &= \text{ARQQ} \\ F_{\text{Vigenere}}\,(\text{ON}, \text{ENCENDIDO}) &= \text{SAQRBQWQC} \end{aligned}$$

Para entender mejor el proceso veamos otro ejemplo en que encriptamos las primeras letras del abecedario usando como contraseña la palabra CLAVE:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
+	C	L	A	V	E	C	L	A	V	E	C	L	A	V	E	C	L	A	V	E
	C	M	C	Y	I	H	R	H	D	N	M	W	M	I	S	R	B	R	N	X

Hace falta entender que sumar dos letras equivale a transformar ambas letras a enteros y dar la letra que corresponda al entero obtenido de sumar los dos anteriores. De esta forma el proceso de desencriptado es análogo pero con el operador resta, siempre en módulo 26:

	C	M	C	Y	I	H	R	H	D	N	M	W	M	I	S	R	B	R	N	X
−	C	L	A	V	E	C	L	A	V	E	C	L	A	V	E	C	L	A	V	E
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T

1.5.3. Implementación

Para poder implementar el criptosistema necesitaremos de una función auxiliar que lo que haga sea crear una cadena infinita repitiendo la palabra clave. Esta función la hemos implementado así:

```
repite :: String -> String
repite = concat . repeat
```

Gracias a la función anterior basta ir asociando cada letra del mensaje con la de la cadena infinita y sumar. El código en Haskell de la función encriptadora F que recibe como datos la clave k, que es una cadena de caracteres; y el mensaje m, que es otra cadena de caracteres es:

```
vigenereF :: String -> String -> String
vigenereF k m = map (\x -> int2char (x `mod` 26)) (zipWith (+) xs ys)
    where xs = str2int m
          ys = str2int (repite k)
```

Y la función desencriptadora G que recibe un mensaje codificado y la clave con el que se encriptó, y nos devuelve el mensaje original se implementa de la siguiente forma:

```
vigenereG :: String -> String -> String
vigenereG k m' = map (\x -> int2char (x `mod` 26)) (zipWith (-) xs ys)
    where xs = str2int m'
          ys = str2int (repite k)
```

1.5.4. Corrección

En esta sección tratamos de comprobar la corrección de la implementación que hemos hecho de las funciones *F* y *G* del código Vigenère. Para ello, tenemos que definir un generador aleatorio de mensajes y de claves. En este caso, como tanto el mensaje como la clave son cadenas de caracteres implementaremos un único generador que nos sirva para los dos. Nos apoyaremos de la función `genLetra` de la página 23:

```
mensajeOclaveV :: Gen String
mensajeOclaveV = listOf1 genLetra
```

La propiedad de que al encriptar y desencriptar un mensaje con la misma clave obtenemos el mensaje original se implementa así:

```
prop_CorreccionVigenere :: Property
prop_CorreccionVigenere =
    forAll mensajeOclaveV
           (\k -> forAll mensajeOclaveV
                         (\m -> m == vigenereG k (vigenereF k m)))
```

Que al ejecutar en la consola:

```
ghci> quickCheck prop_CorreccionVigenere
+++ OK, passed 100 tests.
```

Y por tanto podemos aumentar la confianza en que las funciones encriptadora y desencriptadora funcionan como deseamos.

1.6. Criptosistema de Hill

1.6.1. Introducción

El **cifrado de Hill** es un criptosistema polialfabético que trabaja dividiendo el mensaje original en bloques de una tamaño fijo y tranformando cada bloque de forma independiente en otro conjunto de letras distinto. Esta transformación viene definida por una aplicación del álgebra lineal: la multiplicación matricial.

Este sistema es polialfabético pues puede darse que un mismo carácter en un mensaje a enviar se encripte en dos caracteres distintos en el mensaje encriptado.

Lester Hill trató por primera vez este criptosistema en 1929 en *The American Mathematical Monthly*, introduciendo así uno de las primeras aplicaciones del álgebra lineal a la criptografía poligráfica. En 1931, volvió a escribir un artículo sobre el cifrado en otra edición del mismo periódico.

Hill, con ayuda de Louis Weisner, tuvieron la idea de construir una máquina que implementase el criptosistema. La llamaron *the Message Protector* y la patentaron. La máquina operaba con bloques de seis letras y se basaba en un sistema de engranajes y poleas.

1.6.2. Descripción matemática

Sea un mensaje $m = m_1 m_2 ...$ donde cada m_i es el entero correspondiente a la letra i–ésima del mensaje. Y sea la clave dada por una matriz cuadrada K de dimensión n. El procedimiento de cifrado consiste en dividir nuestro mensaje en vectores de longitud n (denotados por v_j), para, posteriormente, multiplicar cada vector por nuestra matriz clave. Por tanto, la función encriptadora F se define:

$$F_{Hill}(v_j, K) = K \times v_j = m'_j$$

El proceso de desencriptado es análogo, cambiando la matriz K por su inversa módulo 26. Así que la función desencriptadora G se define:

$$G_{Hill}(m'_j, K) = K^{-1} \times m'_j = v_j$$

Hay que tener en cuenta que los elementos de la matriz K son de $\mathbb{F}_{26}$ y que siempre operamos en módulo 26. Por tanto, la matriz K debe ser invertible para que el criptosistema funcione correctamente.

Veamos un ejemplo de cómo se encripta: Sea m el mensaje "MATEMATICAS" cuyo código es $(12, 0, 19, 4, 12, 0, 19, 8, 2, 0, 18)$ y sea $K = \begin{pmatrix} 2 & 1 \\ 3 & 4 \end{pmatrix}$

$$F_{Hill}\left(\begin{pmatrix}12\\0\end{pmatrix},\begin{pmatrix}2&1\\3&4\end{pmatrix}\right) = \begin{pmatrix}2&1\\3&4\end{pmatrix}\times\begin{pmatrix}12\\0\end{pmatrix} = \begin{pmatrix}24\\10\end{pmatrix}$$

$$F_{Hill}\left(\begin{pmatrix}19\\4\end{pmatrix},\begin{pmatrix}2&1\\3&4\end{pmatrix}\right) = \begin{pmatrix}2&1\\3&4\end{pmatrix}\times\begin{pmatrix}19\\4\end{pmatrix} = \begin{pmatrix}16\\21\end{pmatrix}$$

Y así sucesivamente. Finalmente, se obtiene

$$F_{Hill}\left(MATEMATICAS,\begin{pmatrix}2&1\\3&4\end{pmatrix}\right) = YKQVYKULEGS$$

1.6.3. Implementación

Como hemos comentado anteriormente, para poder encriptar y desencriptar correctamente no nos sirve cualquier matriz clave. El primer paso será implementar una función que verifique si la matriz es invertible módulo 26. El código queda:

```
invertible :: Matrix Int -> Bool
invertible a = 1 == gcd x 26
    where x = abs (detLaplace a)
```

Por ejemplo,

```
ghci> invertible (fromLists [[2,1],[3,4]])
True
ghci> invertible (fromLists [[2,1],[8,4]])
False
```

De esta forma, ya podemos definir la función encriptadora F:

```
hillF :: Matrix Int -> String -> String
hillF ks m | invertible ks = aux ks m
           | otherwise     = error "Matriz no valida"
  where
    aux ks [] = []
    aux ks m
      | nrows (ks) <= length m =
          (int2str
           (toList
            (mapCol
             (\_ x -> x `mod` 26)
             1
```

```
            (multStd ks
                     (transpose
                      (fromLists [str2int (take (nrows ks) m)])))))
        ++ aux ks (drop (nrows (ks)) m)
    | otherwise = m
```

Para poder implementar la función desencriptadora necesitamos antes unos pasos previos, ya que debemos calcular la matriz inversa en módulo 26. Comenzaremos por implementar la matriz adjunta:

```
matrizAdj :: Matrix Int -> Matrix Int
matrizAdj a =
    matrix (nrows a)
           (ncols a)
           \ (i,j) -> (((-1)^(i+j)) * detLaplace (minorMatrix i j a))
```

El siguiente paso es trasponer la matriz adjunta:

```
matrizAdjTrasp :: Matrix Int -> Matrix Int
matrizAdjTrasp = transpose . matrizAdj
```

Usando las dos funciones anteriores y la función `invMod` definida en la página 28 ya podemos implementar el cálculo de la matriz inversa:

```
matrizInv :: Matrix Int -> Matrix Int
matrizInv a
    | invertible a = matrizInvAux (matrizAdjTrasp a) (nrows a)
    | otherwise    = error "Matriz no valida"
    where
      matrizInvAux b 0 = b
      matrizInvAux b n =
          matrizInvAux (mapRow (\_ x -> ((x * t) `mod` 26)) n b) (n-1)
      t = invMod (detLaplace a `mod`26) 26
```

Veamos unos ejemplos de cómo actúan estas tres funciones:

```
ghci> matrizAdj (fromLists [[2,3],[1,1]])
(  1 -1 )
( -3  2 )
ghci>  matrizAdjTrasp (fromLists [[2,3],[1,1]])
```

```
(  1 -3 )
( -1  2 )
ghci> matrizInv (fromLists [[2,3],[1,1]])
( 25  3 )
(  1 24 )
ghci> matrizInv (fromLists [[2,3],[1,7]])
(  3 21 )
(  7 12 )
```

Finalmente, ya es sencillo implementar la función desencriptadora. El código es:

```
hillG :: Matrix Int -> String -> String
hillG ks = hillF (matrizInv ks)
```

Veamos ahora a las funciones encriptadora y desencriptadora en acción:

```
ghci> hillF (fromLists [[2,1],[3,4]]) "LASMATEMATICASSONLAMUSICADELARAZON"
"WHWYTYUITYSGSUYGLFMWGCSGDMTERQZWPQ"

ghci> hillG (fromLists [[2,1],[3,4]]) "WHWYTYUITYSGSUYGLFMWGCSGDMTERQZWPQ"
"LASMATEMATICASSONLAMUSICADELARAZON"
```

1.6.4. Corrección

Para ver la corrección del criptosistema debemos implementar un generador de mensajes y claves. Para los mensajes usaremos "mensaje" de la página 23. Definimos como sigue un generador de matrices:

```
matriz :: Gen (Matrix Int)
matriz = do n <- choose (2,6)
            xs <- suchThat (vector (n*n)) (esInvertible n)
            return (fromList n n xs)
   where esInvertible n xs = invertible (fromList n n xs)
```

Y unificamos todo en un único generador:

```
mensajeYclaveHill :: Gen (String,Matrix Int)
mensajeYclaveHill = do m <- mensaje
                       k <- matriz
                       return (m,k)
```

Que, por ejemplo, genera las siguientes parejas de mensajes y matrices claves:

```
ghci>  generate mensajeYclaveHill
("KCAFYAXDITGTZFXSKVFEASP",
( 19  6 )
( 10 -7 ))
ghci>  generate mensajeYclaveHill
("JFWQERU",
(  28 -26 -15  20  11 )
(  29  25   8 -16 -32 )
( -30   3  -7 -26  25 )
( -20 -26   9  25   2 )
(  17   2  11   0 -32 ))
ghci>  generate mensajeYclaveHill
("CQCXIMBDP",
(  22 -32  14  -9  19 )
( -22 -20   3  14  -6 )
(  21 -13 -28  -3  28 )
( -27   0  16 -23  31 )
(  12  -8 -19  30  17 ))
```

Por último, definimos la propiedad de que al encriptar un mensaje con `hillF` y posteriormente desencriptarlo, obtenemos el original:

```
prop_CorreccionHill :: Property
prop_CorreccionHill =
    forAll mensajeYclaveHill
           (\ (m,k) -> m == hillG k (hillF k m))
```

Que al ejecutar en la consola:

```
ghci> quickCheck prop_CorreccionHill
+++ OK, passed 100 tests.
```

1.7. Criptosistema por Permutación

1.7.1. Introducción

El **cifrado por Permutación**, o **cifrado por trasposición**, es un criptosistema que trabaja, como en casos anteriores, dividiendo el mensaje original el bloques de tamaño fijo y transformando cada bloque de forma independiente. Esta transformación, como su propio nombre indica, es una permutacion de las letras.

Este cifrado era muy usado en la criptografía clásica. Y, normalmente, estas permutaciones se basaban en diseños geométricos.

Un ejemplo de cifrado por trasposición es la Escítala, usado por los espartanos para enviar sus mensajes en época de guerra. Necesitaba de dos varas con forma de prisma del mismo grosor y una cinta de cuero donde se escribía el mensaje. El cifrado consistía en enrollar la cinta alrededor de la vara, escribir el mensaje verticalmente y enviar la cinta. El receptor (que tenía la otra vara del mismo grosor) sólo tenía que enrollar la cinta y leer el mensaje. Como el lector habrá notado, en realidad es, simplemente, una permutación de la misma longitud que el texto a encriptar.

1.7.2. Descripción matemática

Sea un mensaje $m = m_1 m_2 ...$ donde cada m_i es el entero correspondiente a la letra i–ésima del mensaje. Y sea la clave dada por una permutación P de dimensión n. El procedimiento de cifrado consiste en dividir nuestro mensaje en vectores de longitud n (denotados por v_j), para, posteriormente, aplicarle a cada vector la permutación clave. Por tanto, la función encriptadora F se define:

$$F_{Perm}(v_j, P) = P(v_j) = v'_j$$

El proceso de desencriptado es análogo, cambiando la permutación por su inversa. Quedando la función desencriptadora G definida de la siguiente forma:

$$G_{Perm}(v'_j, P) = P^{-1}(v'_j) = v_j$$

1.7.3. Implementación

Para poder implementar este criptosistema debemos definir una función que aplique una permutación dada a una lista. Lo haremos en dos pasos. Definimos la función `ordena` que lo que hace es ordenar los segundos elementos de una lista de pares en función de los primeros:

```
ordena :: Ord a => [(a,b)] -> [b]
ordena [] = []
```

```
ordena ((n,x):xs) = ordena anteriores ++ x : ordena posteriores
    where anteriores  = [(m,y) | (m,y) <- xs, m <= n]
          posteriores = [(k,z) | (k,z) <- xs, k > n]
```

Quedando la función que aplica una permutación implementada así:

```
permuta :: [Int] -> [a] -> [a]
permuta ns xs = ordena (zip ns xs)
```

Unos ejemplos de cómo actúa esta función son:

```
ghci> permuta [2,1,3] "abc"
"bac"
ghci> permuta [4,3,2,1] "abcd"
"dcba"
```

Ya estamos en disposición de cifrar mensajes con este criptosistema. La función encriptadora F queda implementada de la siguiente forma:

```
permutacionF :: [Int] -> String -> String
permutacionF _  [] = []
permutacionF ks m
    | length ks <= length m =
        permuta ks (take (length ks) m)
        ++ permutacionF ks (drop (length ks) m)
    | otherwise = m
```

Para poder descifrar el mensaje necesitamos poder calcular la permutación inversa de una dada:

```
permutaInv :: [Int] -> [Int]
permutaInv ns = permuta ns [1..]
```

De esta forma, implementamos la función desencriptadora:

```
permutacionG :: [Int] -> String -> String
permutacionG ks = permutacionF (permutaInv ks)
```

Veamos un ejemplo de mensajes encriptado y desencriptado con este criptosistema:

```
ghci> permutacionF [1,3,4,2] "ELALGEBRAESGENEROSAAMENUDODAMASDELOQUESELEPIDE"
"ELLAGREBAGESERNEOASAMUENDAODMDASEQLOUEESLIEPDE"

ghci> permutacionG [1,3,4,2] "ELLAGREBAGESERNEOASAMUENDAODMDASEQLOUEESLIEPDE"
"ELALGEBRAESGENEROSAAMENUDODAMASDELOQUESELEPIDE"
```

1.7.4. Corrección

Para ver la corrección del criptosistema debemos implementar un generador de mensajes y claves. Para los mensajes usaremos "mensaje" de la página 23. Definimos como sigue un generador de permutaciones:

```
permutacion :: Gen [Int]
permutacion = do n <- choose (2,6)
                 shuffle [1..n]
```

Ahora lo unificamos todo en un único generador:

```
mensajeYclavePerm :: Gen (String,[Int])
mensajeYclavePerm = do m <- mensaje
                       k <- permutacion
                       return (m,k)
```

Implementamos la propiedad de que desencriptar un mensaje previamente encriptado con estas funciones nos devuelve el mensaje original de la siguiente forma:

```
prop_CorreccionPerm :: Property
prop_CorreccionPerm =
    forAll mensajeYclavePerm
           (\ (m,k)-> m == permutacionG k (permutacionF k m))
```

Y la comprobación en la consola queda:

```
ghci> quickCheck prop_CorreccionPerm
+++ OK, passed 100 tests.
```

Y por tanto podemos aumentar la confianza en que las funciones encriptadora y desencriptadora funcionan como deseamos.

1.8. Criptosistema DES

1.8.1. Introducción

El **cifrado DES** (en inglés, "Data Encryption Standard") fue uno de los algoritmos de encriptación más usados en el mundo de la electrónica, y su influencia fue fundamental en el desarrollo de la criptografía moderna en el mundo académico.

Basándose en el trabajo de Horst Feistel, la empresa estadounidense IBM desarrolló este criptosistema. Y, en 1976, la NBS (National Boreau of Standards) eligió una pequeña modificación hecha por la NSA (National Security Agency) como algoritmo de encriptado para proteger material electrónico clasificado para el gobierno de los E.E.U.U. En 1977 se publicó como un FIPS (Federal Information Processing Standard) oficial de los Estados Unidos.

Actualmente, se considera al DES como inseguro, debido al pequeño tamaño de las claves que utiliza. Sin embargo, el triple DES sí que se considera seguro en la práctica. En los últimos años, el cifrado ha sido sustituido por el AES (Advanced Encryption Standard).

1.8.2. Descripción matemática e implementación

Dividiremos la implementación en tres pasos: Primero, definir el tipo de permutaciones que vamos a usar; segundo, crear 16 sub–claves; y finalmente, el cifrado en sí.

Expansiones

Antes de nada, debemos saber que aplicaremos unas permutaciones a los bits del mensaje. Así que primero definiremos estas permutaciones, que son algo distintas a las de la sección anterior, y para diferenciarlas las denominaremos expansiones. En definitiva, iremos sustituyendo cada número de la expansión por el que ocupa dicha posición en la lista original. De esta forma, la definición por recursión queda:

```
expansionR :: [Int] -> [a] -> [a]
expansionR [] _      = []
expansionR (e:xs) ys = (ys!!(e-1)) : expansionR xs ys
```

Por ejemplo,

```
expansionR [2,1,3]   "abc"         == "bac"
expansionR [4,1,3,2] "abcd"        == "dacb"
expansionR [1,1,4,1,3,2,1] "abcd" == "aadacba"
```

Una definición alternativa usando plegados es:

```
expansion :: [Int] -> [a] -> [a]
expansion xs ys = foldr (\ x y-> (ys!!(x-1)):y) [] xs
```

Comprobemos ahora que ambas definiciones son equivalentes. Para ello nos hace falta implementar un generador de expansiones. Éste generará una permutacion y la duplicará (para que la longitud del expandido sea mayor).

```
expansionGen :: [a] -> Gen [Int]
expansionGen xs = do k <- shuffle [1..length xs]
                     return (k++k)
```

Por ejemplo,

```
ghci> generate (expansionGen "ab")
[1,2,1,2]
ghci> generate (expansionGen "ab")
[2,1,2,1]
```

Quedando la propiedad implementada de la siguiente forma:

```
prop_Expansion :: String -> Property
prop_Expansion xs =
    forAll (expansionGen xs)
    (\ p -> expansionR p xs == expansion p xs)
```

Y al ejecutar:

```
ghci> quickCheck prop_Expansion
+++ OK, passed 100 tests.
```

Sólo nos queda elegir la que sea más eficiente:

```
ghci> :set +s
ghci>  length (expansionR [1..(10^7)] (replicate (10^7) ('a')))
10000000
(4.42 secs, 2563634096 bytes)
ghci>  length (expansion [1..(10^7)] (replicate (10^7) ('a')))
10000000
(1.66 secs, 1843532320 bytes)
```

Por tanto, nos quedamos con la definición por plegados.

Crear sub–claves

Para facilitar la comprensión del lector iremos siguiendo el proceso de crear las 16 sub claves a partir de una en concreto, que llamamos `ejemploK`:

```
ejemploK :: [Int]
ejemploK = [0,0,0,1,0,0,1,1,0,0,1,1,0,1,0,0,0,1,0,1,0,1,1,1,0,1,1,1,1,0,0,
            1,1,0,0,1,1,0,1,1,1,0,1,1,1,1,0,0,1,1,0,1,1,1,1,1,1,1,1,1,0,0,
            0,1]
```

La longitud de la clave en el cifrado DES es de 56 bits, pero la almacenamos en 64 bits (sin usar el octavo de cada byte).

Ahora crearemos, a partir de la clave K, 16–subclaves de 48 bits cada una. Para ello primero aplicaremos una expansión (PC_1) que eliminará los bits de paridad.

```
pc_1 :: [Int]
pc_1 = [57,49,41,33,25,17,9,1,58,50,42,34,26,18,10,2,59,51,43,35,27,19,11,
        3,60,52,44,36,63,55,47,39,31,23,15,7,62,54,46,38,30,22,14,6,61,53,
        45,37,29,21,13,5,28,20,12,4]
```

Que al aplicársela a `ejemploK` nos queda:

```
ghci> expansion pc_1 ejemploK
[1,1,1,1,0,0,0,0,1,1,0,0,1,1,0,0,1,0,1,0,1,0,1,0,1,1,1,1,0,1,0,1,0,1,0,1,
0,1,1,0,0,1,1,0,0,1,1,1,1,0,0,0,1,1,1,1]
```

Posteriormente dividiremos el resultado en dos mitades y operaremos con cada una de ellas de forma independiente. Implementemos una función que haga esto:

```
mitades :: [a] -> ([a],[a])
mitades xs = splitAt (length xs `div` 2) xs
```

Por ejemplo, si dividimos lo que teníamos:

```
ghci> mitades it
([1,1,1,1,0,0,0,0,1,1,0,0,1,1,0,0,1,0,1,0,1,0,1,0,1,1,1,1],
 [0,1,0,1,0,1,0,1,0,1,1,0,0,1,1,0,0,1,1,1,1,0,0,0,1,1,1,1])
```

Otros ejemplos más ilustrativos de cómo funciona mitades son:

```
mitades [1..4]  ==  ([1,2],[3,4])
mitades [1..5]  ==  ([1,2],[3,4,5])
```

Ahora cada mitad la desplazamos una posición a la izquierda. De forma que el bit que ocupa el lugar n pasa a ocupar la posición $n-1$. Observemos que el primer elemento de la lista pasa a la última posición. El código es:

```
desplazaIzq :: Int -> [Int] -> [Int]
desplazaIzq n xs = drop n xs ++ take n xs
```

Al aplicarle un desplazamiento a lo que teníamos de nuestro ejemplo:

```
(desplazaIzq 1 (fst it),desplazaIzq 1 (snd it))
([1,1,1,0,0,0,0,1,1,0,0,1,1,0,0,1,0,1,0,1,0,1,0,1,1,1,1,1],
 [1,0,1,0,1,0,1,0,1,1,0,0,1,1,0,0,1,1,1,1,0,0,0,1,1,1,1,0])
```

Mientras que otros ejemplos más simples del uso de `desplazaIzq` son:

```
desplazaIzq 1 [1,2,3]   == [2,3,1]
desplazaIzq 2 [1,2,3,4] == [3,4,1,2]
```

El siguiente paso es usar la función `desplazaIzq` 16 veces, siguiendo el orden establecido por la lista desplazamientos, para generar 16 sub-claves:

```
desplazamientos :: [Int]
desplazamientos = [1,2,4,6,8,10,12,14,15,17,19,21,23,25,27,28]
```

Finalmente sólo queda aplicarle a cada una de las 16 sub–claves una permutacion (PC_2) aquí definida:

```
pc_2 :: [Int]
pc_2 = [14,17,11,24,1,5,3,28,15,6,21,10,23,19,12,4,26,8,16,7,27,20,13,2,41,
        52,31,37,47,55,30,40,51,45,33,48,44,49,39,56,34,53,46,42,50,36,29,
        32]
```

Ahora implementaremos una función que llamaremos `paso1` que creará las dieciséis sub–claves siguiendo el todo el proceso que hemos detallado más arriba:

```
crea16Claves :: [Int] -> [[Int]]
crea16Claves ks = [expansion pc_2 k | k <- ks'']
    where (xs,ys) = mitades ks'
          ks'     = expansion pc_1 ks
          ks''    = [desplazaIzq n xs ++ desplazaIzq n ys
                     | n <- desplazamientos]
```

Que al aplicarla sobre nuestro `ejemploK` de la página 45 nos quedan las sub–claves:

```
ghci> paso1 ejemploK
[[0,0,0,1,1,0,1,1,0,0,0,0,0,0,1,0,1,1,1,0,1,1,1,1,1,1,1,1,1,0,0,0,1,1,1,
  0,0,0,0,0,1,1,1,0,0,1,0],[0,1,1,1,1,0,0,1,1,0,1,0,1,1,1,0,1,1,0,1,1,0,0,
  1,1,1,0,1,1,0,1,1,1,1,1,0,0,1,0,0,1,1,1,1,1,0,0,1,0,1], [0,1,0,1,0,1,0,1,1,
  1,1,1,1,1,0,0,1,0,0,0,1,0,1,0,0,1,0,0,0,0,1,0,1,1,0,0,1,1,1,1,1,0,0,1,1,
  0,0,1],[0,1,1,1,1,0,0,1,0,1,0,1,0,1,1,0,1,1,1,0,1,0,1,1,0,1,1,0,1,1,0,1,1,
  0,0,1,1,0,1,0,1,0,0,0,1,1,1,0,1],[0,1,1,1,1,1,1,0,0,1,1,1,0,1,1,0,0,0,0,0,
  0,0,1,1,1,1,1,1,1,0,1,0,1,1,0,1,0,1,0,0,1,1,1,0,1,0,1,0,0,0],[0,1,1,0,0,0,
  1,1,1,0,1,0,0,1,0,1,0,0,1,1,1,1,1,0,0,1,0,1,0,0,0,0,0,1,1,1,1,0,1,1,0,0,
  1,0,1,1,1,1],[1,1,1,0,1,1,0,0,1,0,0,0,0,1,0,0,1,0,1,1,0,1,1,1,1,1,1,1,1,0,
  1,1,0,0,0,0,1,1,0,0,0,1,0,1,1,1,1,1,0,0],[1,1,1,1,0,1,1,1,1,1,0,0,0,1,0,1,0,
  0,0,1,1,1,0,1,0,1,1,0,0,0,0,0,0,1,0,0,1,1,1,0,1,1,1,1,1,1,1,1,0,1,1],[1,1,1,
  0,0,0,0,0,1,1,0,1,1,0,1,1,1,1,1,1,0,1,0,1,1,1,1,1,1,0,1,1,0,1,1,1,1,1,0,0,1,1,
  1,1,0,0,0,0,0,0,0,1],[1,0,1,1,0,0,0,1,1,1,1,1,1,0,0,1,1,0,1,0,0,0,1,1,1,1,1,0,
  1,1,1,0,1,0,0,1,0,0,0,1,1,0,0,1,0,0,1,1,1,1],[0,0,1,0,0,0,0,1,0,1,0,1,1,
  1,1,1,1,1,0,1,0,0,1,1,1,1,1,0,1,1,1,1,0,1,1,0,1,0,0,1,1,1,0,0,0,0,1,1,0],
  [0,1,1,1,0,1,0,1,0,1,1,1,0,0,0,1,1,1,1,1,0,1,0,1,1,0,0,1,0,1,0,0,0,1,1,
  0,0,1,1,1,1,1,1,0,1,0,0,1],[1,0,0,1,0,1,1,1,1,1,0,0,0,1,0,1,1,1,0,1,0,0,
  0,1,1,1,1,1,1,1,0,1,0,1,0,1,1,1,0,1,0,0,1,0,0,0,0,0,1],[0,1,0,1,1,1,1,1,1,0,
  1,0,0,0,0,1,1,1,0,1,1,0,1,1,1,1,1,1,1,1,0,0,1,0,1,1,1,0,0,1,1,1,0,0,1,1,1,
  0,1,0],[1,0,1,1,1,1,1,1,1,1,0,0,1,0,0,0,1,1,0,0,0,1,1,0,1,0,0,1,1,1,1,1,0,1,
  0,0,1,1,1,1,1,1,1,0,0,0,0,1,0,1,0],[1,1,0,0,1,0,1,1,0,0,1,1,1,1,0,1,1,0,0,
  0,1,0,1,1,0,0,0,0,1,1,1,0,0,0,0,1,0,1,1,1,1,1,1,1,0,1,0,1]]
```

Rondas de Feistel

Una parte esencial del criptosistema DES son las denominadas **rondas de Feistel** o **redes de Feistel**, que no son más que un tipo de cifrado en bloque con una estructura muy particular. Tiene la ventaja de que las operaciones de cifrado y descifrado son idénticas, únicamente hace falta invertir el orden de las claves utilizadas.

Análogamente a la sección anterior, sabemos que el mensaje de 64 bits se puede dividir en dos mitades de 32 bits cada una ($M = [L|R]$).

Además, para definir la ronda de Feistel usaremos una función $f(R,k)$ que recibe un bloque de 32 bits (R) y una de las sub–claves de 48 bits (k) y nos devuelve un bloque de 32 bits. Una ronda de Feistel queda definida de la siguiente forma:

$$\begin{cases} L = R \\ R = L \oplus f(R,k) \end{cases}$$

En definitiva, con cada iteración, los 32 bits que estaban más a la derecha pasan a ocupar el lado de la izquierda, y los que estaban a la izquierda los sumamos con el

operador xor al resultado de la función f.

Con la idea de facilitar la comprensión fijamos un mensaje ($M = ejemploL|ejemploR$) y una clave (`ejemplok`) sobre las que iremos trabajando:

```
ejemploL, ejemploR, ejemplok :: [Int]
ejemploL =
    [1,0,0,0,0,0,1,0,0,1,0,0,0,0,1,0,1,1,0,0,0,0,1,0,1,0,1,0,0,0,1,0]
ejemploR =
    [0,0,1,0,0,0,1,0,0,1,1,0,0,0,1,0,1,1,1,0,0,0,1,0,0,0,0,1,0,0,1,0]
ejemplok =
    [0,0,0,1,1,0,1,1,0,0,0,0,0,0,1,0,1,1,1,0,1,1,1,1,1,1,1,1,1,1,0,
     0,0,1,1,1,0,0,0,0,0,1,1,1,0,0,1,0]
```

Para poder implementar las rondas de Feistel comenzaremos implementando la función $f(R,k)$, que consta de varios pasos:

1. Realizamos una expansión de R definida por :

```
ex :: [Int]
ex = [32,1,2,3,4,5,4,5,6,7,8,9,8,9,10,11,12,13,12,13,14,15,16,17,16,17,
     18,19,20,21,20,21,22,23,24,25,24,25,26,27,28,29,28,29,30,31,32,1]
```

   En nuestro ejemplo nos queda:

```
ghci> expansion ex ejemploR
[0,0,0,1,0,0,0,0,0,1,0,0,0,0,1,1,0,0,0,0,0,1,0,1,0,1,1,1,0,0,0,0,0,1,0,
 0,0,0,0,0,1,0,1,0,0,1,0,0]
```

2. La siguiente tarea será sumar mediante la función xor la expansion de R y la clave k:

```
xor :: [Int] -> [Int] -> [Int]
xor xs ys = [xor' x y | (x,y) <- zip xs ys]
            where xor' x y = bool2int (x /= y)
```

   Que siguiendo el ejemplo nos da:

```
ghci> xor it ejemplok
[0,0,0,0,1,0,1,1,0,1,0,0,0,0,0,1,1,1,1,0,1,0,1,0,1,0,0,0,1,1,0,0,0,0,1,
 1,0,0,0,0,1,1,0,1,0,1,1,0]
```

3. Lo obtenido lo dividiremos en 8 listas de 6 bits cada una:

```
divideEn8 :: [Int] -> [[Int]]
divideEn8 [] = []
divideEn8 xs = take 6 xs : divideEn8 (drop 6 xs)
```

Al dividir lo que teniamos del ejemplo obtenemos:

```
ghci> divideEn8 it
[[0,0,0,0,1,0],[1,1,0,1,0,0],[0,0,0,1,1,1],[1,0,1,0,1,0],[1,0,0,0,1,1],
 [0,0,0,0,1,1],[0,0,0,0,1,1],[0,1,0,1,1,0]]
```

4. Haremos "pasar" cada lista por una de las S-cajas. El listado de las ocho S–cajas es:

```
s1, s2, s3, s4, s5, s6, s7, s8 :: Matrix Int
s1 = fromLists [[14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7],
                [0,15,7,4,14,2,13,1,10,6,12,11,9,5,3,8],
                [4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0],
                [15,12,8,2,4,9,1,7,5,11,3,14,10,0,6,13]]

s2 = fromLists [[15,1,8,14,6,11,3,4,9,7,2,13,12,0,5,10],
                [3,13,4,7,15,2,8,14,12,0,1,10,6,9,11,5],
                [0,14,7,11,10,4,13,1,5,8,12,6,9,3,2,15],
                [13,8,10,1,3,15,4,2,11,6,7,12,0,5,14,9]]

s3 = fromLists [[10,0,9,14,6,3,15,5,1,13,12,7,11,4,2,8],
                [13,7,0,9,3,4,6,10,2,8,5,14,12,11,15,1],
                [13,6,4,9,8,15,3,0,11,1,2,12,5,10,14,7],
                [1,10,13,0,6,9,8,7,4,15,14,3,11,5,2,12]]

s4 = fromLists [[7,13,14,3,0,6,9,10,1,2,8,5,11,12,4,15],
                [13,8,11,5,6,15,0,3,4,7,2,12,1,10,14,9],
                [10,6,9,0,12,11,7,13,15,1,3,14,5,2,8,4],
                [3,15,0,6,10,1,13,8,9,4,5,11,12,7,2,14]]

s5 = fromLists [[2,12,4,1,7,10,11,6,8,5,3,15,13,0,14,9],
                [14,11,2,12,4,7,13,1,5,0,15,10,3,9,8,6],
                [4,2,1,11,10,13,7,8,15,9,12,5,6,3,0,14],
                [11,8,12,7,1,14,2,13,6,15,0,9,10,4,5,3]]

s6 = fromLists [[12,1,10,15,9,2,6,8,0,13,3,4,14,7,5,11],
                [10,15,4,2,7,12,9,5,6,1,13,14,0,11,3,8],
```

```
                [9,14,15,5,2,8,12,3,7,0,4,10,1,13,11,6],
                [4,3,2,12,9,5,15,10,11,14,1,7,6,0,8,13]]

s7 = fromLists [[4,11,2,14,15,0,8,13,3,12,9,7,5,10,6,1],
                [13,0,11,7,4,9,1,10,14,3,5,12,2,15,8,6],
                [1,4,11,13,12,3,7,14,10,15,6,8,0,5,9,2],
                [6,11,13,8,1,4,10,7,9,5,0,15,14,2,3,12]]

s8 = fromLists [[13,2,8,4,6,15,11,1,10,9,3,14,5,0,12,7],
                [1,15,13,8,10,3,7,4,12,5,6,11,0,14,9,2],
                [7,11,4,1,9,12,14,2,0,6,10,13,15,3,5,8],
                [2,1,14,7,4,10,8,13,15,12,9,0,3,5,6,11]]
```

Las S–cajas son matrices de las que extraeremos un número de la siguiente forma:

a) El primer y el último bit del bloque, en base 2, representan la fila.

b) los 4 bits restantes (en el centro) representan la columna

La función que recibe un par (bloque,Scaja) y al bloque de longitud 6 le aplica la S–caja es:

```
sCaja :: ([Int],Matrix Int) -> [Int]
sCaja ([a,b,c,d,e,f],s) = int2bin4 (s!(i,j))
    where i = 1 + bin2int4 [a,f]
          j = 1 + bin2int4 [b,c,d,e]
```

Aplicando esta función al primer bloque y con la primera S–caja:

```
ghci> sCaja (head it,s1)
[0,1,0,0]
```

5. El último paso para calcular la f es concatenar lo obtenido de las S–cajas y aplicarle una expansión definida por:

```
p :: [Int]
p = [16,7,20,21,29,12,28,17,1,15,23,26,5,18,31,10,2,8,24,14,32,27,3,9,
     19,13,30,6,22,11,4,25]
```

Por tanto, aglutinando todo el proceso, la implementación de la función f nos queda:

```
funcionF :: [Int] -> [Int] -> [Int]
funcionF k r =
    expansion p (concat [sCaja a
                         | a <-zip (divideEn8 (xor (expansion ex r) k))
                                   [s1,s2,s3,s4,s5,s6,s7,s8]])
```

Quedando $f(ejemploR, ejemplok)$ de la siguiente forma:

```
ghci> funcionF ejemplok ejemploR
[1,0,0,1,1,1,0,1,0,1,1,0,1,0,1,0,1,0,1,0,0,0,0,1,0,1,1,1,1,0,0,0]
```

Una vez implementada la función f es sencillo ahora escribir el código de una ronda de Feistel, ayudándonos de la funcion mitades de la página 45:

```
rondaFeistel :: [Int] -> [Int] -> [Int]
rondaFeistel k m = r ++ xor l (funcionF k r)
    where (l,r) = mitades m
```

Que en nuestro ejemplo quedaría:

```
ghci> rondaFeistel ejemplok (ejemploL ++ ejemploR)
[0,0,1,0,0,0,1,0,0,1,1,0,0,0,1,0,1,1,1,0,0,0,1,0,0,0,0,1,0,0,1,0,0,0,0,1,
 1,1,1,1,0,0,1,0,1,0,0,0,0,1,1,0,0,0,1,1,1,1,0,1,1,0,1,0]
```

Implementación del encriptador

El cifrado DES opera dividiendo el mensaje M en bloques de texto de 64 bits y cifrando cada bloque de forma independiente. Cada bloque, a su vez se divide por la mitad (32 bits). Llamaremos a la mitad de la iquierda L y a la de la derecha R. Además, el DES usa claves K de un tamaño fijo, 56 bits.

Igual que antes, a medida que escribimos el código iremos encriptando un ejemplo de mensaje ($ejemploM$) con un ejemplo de clave ($ejemploK$):

```
ejemploM :: String
ejemploM = "ABCDEFGH"

ejemploK :: [Int]
ejemploK = [0,0,0,1,0,0,1,1,0,0,1,1,0,1,0,0,0,1,0,1,0,1,1,1,0,1,1,1,1,0,0,
            1,1,0,0,1,1,0,1,1,1,0,1,1,1,1,0,0,1,1,0,1,1,1,1,1,1,1,1,1,0,0,
            0,1]
```

```
codifica "ejemploM"
[1,0,0,0,0,0,1,0,0,1,0,0,0,0,1,0,1,1,0,0,0,0,1,0,0,0,1,0,0,0,1,0,1,0,1,0,0,
 0,1,0,0,1,1,0,0,0,1,0,1,1,1,0,0,0,1,0,0,0,0,1,0,0,1,0]
```

Una vez recibido el mensaje lo primero es aplicarle una expansión inicial (IP)

```
ip :: [Int]
ip = [58,50,42,34,26,18,10,2,60,52,44,36,28,20,12,4,62,54,46,38,30,22,14,
      6,64,56,48,40,32,24,16,8,57,49,41,33,25,17,9,1,59,51,43,35,27,19,11,
      3,61,53,45,37,29,21,13,5,63,55,47,39,31,23,15,7]
```

```
ghci> expansion ip it
[0,1,1,0,0,1,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,
 1,0,1,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1]
```

El siguiente paso será aplicar 16 rondas de Feistel (implementada en la página 51) sucesivamente usando las 16 sub–claves generadas a partir de la clave K (la generación de sub–claves está implementada en la página 46). Ahora daremos el código de la función que aplica 16 veces las rondas de Feistel usando las sub–claves que le corresponda a cada iteración:

```
rondasFeistel :: [Int] -> [Int] -> [Int]
rondasFeistel ks = aux (crea16Claves ks)
    where aux [] m     = snd (mitades m) ++ fst (mitades m)
          aux (k:ks) m = aux ks (rondaFeistel k m)
```

Que al aplicársela a nuestro ejemplo:

```
ghci> rondasFeistel ejemploK it
[0,1,1,0,1,0,0,1,1,1,1,1,0,0,1,1,0,0,1,0,1,0,0,1,1,1,1,1,1,1,0,0,1,1,1,0,
 1,1,1,0,0,0,1,0,1,0,0,1,0,0,1,0,1,0,0,1,0,1,1,1,0,0,0,0]
```

Finalmente deshacemos la permutación IP del principio aplicando su inversa:

```
invIP :: [Int]
invIP =
    [40,8,48,16,56,24,64,32,39,7,47,15,55,23,63,31,38,6,46,14,54,22,62,30,
     37,5,45,13,53,21,61,29,36,4,44,12,52,20,60,28,35,3,43,11,51,19,59,27,
     34,2,42,10,50,18,58,26,33,1,41,9,49,17,57,25]
```

Quedándonos en nuestro ejemplo lo siguiente:

```
ghci> expansion invIP it
[0,1,1,1,1,1,0,0,1,0,0,1,0,0,0,0,1,0,0,0,0,0,0,1,1,1,1,0,1,1,0,1,0,0,0,1,
 0,0,1,1,1,1,1,1,1,1,1,1,1,1,0,1,0,0,1,1,1,0,0,1,0,0,0,1]

ghci> descodifica it
"> t 129 183 200 255 203 137"
```

Una vez que hemos entendido el proceso de encriptación vamos a unir todos los pasos en una sola función que reciba un mensaje y una clave y nos codifique el mensaje con el cifrado DES de clave dada. Debemos de tener en cuenta que el cifrado DES trabaja con mensajes de 64 bits, así que necesitaremos de una función auxiliar antes de dar el código definitivo. La implementación de dicha función quedaría:

```
encriptaDES' :: [Int] -> [Int] -> [Int]
encriptaDES' ks m
    | length m == 64 =
        expansion invIP (rondasFeistel ks (expansion ip m))
    | otherwise = m
```

Como ya hemos comentado anteriormente la clave del cifrado DES es de longitud 56 pero se guarda en 64 bits; por tanto, nosotros supondremos que una clave es una cadena de 8 caracteres que tendremos que pasar a binario. Teniendo esto en cuenta, el código de la función encriptadora es:

```
encriptaDES :: String -> String -> String
encriptaDES ks m =
    descodifica (aux (codifica ks) (codifica m) [])
    where aux k [] xs = xs
          aux k ms xs =
              aux k (drop 64 ms) (xs ++ encriptaDES' k (take 64 ms))
```

Que al aplicárselo a *ejemploM* y *ejemploK* :

```
ghci> encriptaDES (descodifica ejemploK) ejemploM
"> t 129 183 200 255 203 137"
```

Cuando definimos las rondas de Feistel se comentó que aplicarlas sucesivamente tenía un proceso inverso, que no era más que aplicarlas de nuevo invirtiendo el orden de las sub–claves usadas. Así que, definimos una función que aplique las rondas de Feistel en orden inverso:

```
rondasFeistelInv :: [Int] -> [Int] -> [Int]
rondasFeistelInv ks = aux (reverse(crea16Claves ks))
```

```
    where aux [] m     = snd (mitades m) ++ fst (mitades m)
          aux (k:ks) m = aux ks (rondaFeistel k m)
```

Con ayuda de esta función definimos un proceso análogo al de encriptado para desencriptar:

```
desencriptaDES' :: [Int] -> [Int] -> [Int]
desencriptaDES' ks m
    |length m == 64 =
        expansion invIP (rondasFeistelInv ks (expansion ip m))
    | otherwise = m

desencriptaDES :: String -> String -> String
desencriptaDES ks m =
    descodifica (aux (codifica ks) (codifica m) [])
    where aux k [] xs = xs
          aux k ms xs =
              aux k (drop 64 ms) (xs ++ desencriptaDES' k (take 64 ms))
```

Y al intentar desencriptar lo que ya teníamos:

```
ghci> desencriptaDES (descodifica ejemploK) it
"ABCEDFGH"
```

1.8.3. Corrección

En esta sección tratamos de comprobar la corrección de la implementación que hemos hecho de las funciones encriptadora y desencriptadora del criptosistema DES. Para ello, tenemos que definir un generador aleatorio de mensajes y de claves:

```
claveYmensajeDES :: Gen (String,String)
claveYmensajeDES = do n <- choose (1,6)
                      k <- vector 8
                      m <- vector (8*n)
                      return (k,m)
```

Que, por ejemplo, genera el siguiente par:

```
ghci> generate claveYmensajeDES
("253 O w ETX * r V U" , "/ { 228 S Y N 188 n X S V I P : 145 y S I ! ")
```

Definimos ahora la propiedad que comprueba si un mensaje queda invariante al encriptar y posteriormente desencriptar:

```
prop_CorreccionDES :: Property
prop_CorreccionDES =
    forAll claveYmensajeDES
           (\ (k,m) -> m == desencriptaDES k (encriptaDES k m))
```

Al ejecutar con QuickCheck:

```
ghci> quickCheck prop_CorreccionDES
+++ OK, passed 100 tests.
```

Y por tanto podemos aumentar la confianza en que las funciones encriptadora y desencriptadora funcionan como deseamos.

Capítulo 2

Criptografía de clave pública

2.1. Descripción

Durante la mayor parte de la historia de la criptografía una clave se debía mantener en absoluto secreto y se transmitía mendiante métodos seguros pero no criptográficos, como podían ser reuniones cara a cara o un mensajero de confianza. Pero como sabemos, éste método de distribución de claves es muy ineficaz. La criptografía de clave pública se inventa para suplir estas dificultades. Pues con estos protocolos los usuarios se pueden comunicar de manera segura sobre un canal inseguro sin necesidad de acordar una clave anteriormente.

Más formalmente, la **criptografía de clave pública**, también llamada **criptografía asimétrica** o **criptografía de dos claves**, es el método criptográfico que usa un par de claves para el envío de mensajes. Las dos claves pertenecen a la misma persona que ha enviado el mensaje. Una clave es pública y se puede entregar a cualquier persona, la otra es privada y el propietario debe guardarla de modo que nadie tenga acceso a ella.

La primera invención de un algoritmo asimétrico la hicieron *Whitfield Diffie* y *Martin Hellman* en 1976. Estos algoritmos fueron conocidos más tarde como el protocolo de intercambio de claves *Diffie–Hellman* y era un caso particular del *RSA*. Las principales diferencias a tener en cuenta que introdujeron respecto a la criptografía simétrica son:

- Dejamos de pensar en dos usuarios: pensamos en una comunidad de usuarios.
- Dejamos de pensar en una única clave para cifrar y descifrar. En lugar de eso tenemos dos claves y una función trampilla: conociendo la clave de cifrado es sencillo averiguar la de descifrado, pero el camino inverso no es factible computacionalmente.
- Dejamos de pensar en encriptar para que sólo emisor y receptor puedan desencriptar. En lugar de eso, encriptaremos de forma que sólo el destinatario pueda desencriptar.

Desde los años 70, se han desarrollado un gran número y variedad de cifrados, firmas digitales y otras técnicas en el campo de la criptografía de clave pública. En el transcurso de este capítulo trataremos con tres ejemplos muy famosos como son el *RSA*, *ElGamal* y el de *Rabin*. Pero, por supuesto, existen muchísimos más ejemplos, entre los que destaca la familia de algoritmos llamados *de curva elíptica* que están a la vanguardia de la criptografía actual.

En la criptografía de clave pública cabe destacar que el emisor y el receptor no comparten la misma información. Esta diferencia de acceso a la información hace que estos códigos se denominen también *asimétricos*. Es obvio que el diseño de un protocolo de este tipo ha de ser muy cuidadoso para evitar que un ataque alternativo al esperado rompa la seguridad del sistema. Toda la seguridad reacae sobre lo que se conoce como *función trampilla*. Éstas son funciones sencillas de calcular, pero que, sin toda la información son irreversibles computacionalmente. Por tanto, en todos los protocolos sacrificaremos una parte de la información haciéndola pública (clave de encriptación), pero a partir de la cual no se podrá obtener (teóricamente sí, pero computacionalmente no) todo el resto de la información que permanece oculta (clave de desencriptación).

Las funciones trampilla no son sencillas de encontrar. De hecho, en los últimos años, cada problema del que se sabe que es un buen candidato a función tampilla ha sido probado como base para un criptosistema de clave pública. A continuación vamos a ver los tres problemas que dan lugar a los criptosistemas que implementaremos en este capítulo.

La factorización:

Si tomamos dos primos grandes (por ejemplo p y q) cualquier ordenador los multiplica enseguida ($p \cdot q = N$). Sin embargo, dado N, hallar p y q no es tan simple. Esta función trampilla se utiliza como base para el protocolo RSA, uno de los más usados en la actualidad.

El DLP o problema del logaritmo discreto:

Dado un primo p y un natural $g < p$, hallar $h = g^a \mod p$ para un a cualquiera es muy sencillo. Pero dado h hallar el exponente a es imposible (para p grande). Esta función trampilla se usa en el *DHEP* y en un método de encriptación llamado *ElGamal*, que es la base de los dos métodos más usados en internet hoy en día: *PGP* y *GPG*.

El problema de las raíces cuadradas

Dado un entero no primo (n) y otro entero aleatorio menor (g), es sencillo hallar $x = g^2 \mod n$. Pero dado x, hallar g (su raíz cuadrada módulo n) es impracticable. Este problema es la base del método de *Rabin*.

2.2. Primeros pasos

Como ya hemos comentado con anterioridad, en la criptografía, definimos una forma de asociar a cada mensaje con un objeto que sea fácil de transformar. Por ejemplo, en el capítulo anterior, dividimos el mensaje en caracteres y describimos principalmente dos posibilidades de asociarlos: a números enteros ó a números binarios. En cambio, para este capítulo, iremos algo más allá; le asociaremos directamente a cada mensaje un número entero (generalmente muy grande). Al hacer dicha asociación podremos trabajar con este número, en vez de hacerlo con el mensaje tal cual.

2.2.1. Cuarta asociación

Para poder dar el número entero que corresponde a cada mensaje, sustituiremos cada carácter por su número en código ASCII y concatenaremos todos los números. Por ejemplo, si el mensaje es AB, el número que le corresponderá es el 65066; ya que $A = 065$ y $B = 066$. Teniendo esto en cuenta definimos la función que nos asocia una cadena de caracteres a un número entero (grande):

```
strToInteger1 :: String -> Integer
strToInteger1 cs =
    foldl (\x y -> toInteger y + 1000 * toInteger x) 0 (map ord cs)
```

Aunque, aprovechando que los caracteres son un tipo de dato ordenado podemos dar una definición alternativa:

```
strToInteger :: String -> Integer
strToInteger = read . concatMap aux
    where aux c | c < '\n' = "00" ++ cs
                | c < 'd'  = '0' : cs
                | otherwise = cs
                where cs = show (ord c)
```

Veamos unos ejemplos:

```
ghci> strToInteger "A"
65
ghci> strToInteger "AA"
65065
ghci> strToInteger "ABCDEFGHIJ"
650660670680690700710720730074
```

Comparando la eficiencia en la consola vemos que la segunda definición es mejor:

```
ghci> length (show (strToInteger (replicate (10^5) 'A')))
299999
(7.93 secs, 37673797896 bytes)
ghci> length (show (strToInteger1 (replicate (10^5) 'A')))
299999
(25.14 secs, 12549525056 bytes)
```

Aunque, como la diferencia es tan grande, podría ser que no sean exactamente iguales los resultados. Por ello, y para aumentar nuestro grado de confianza en las funciones, implementaremos la siguiente propiedad con el objetivo de poder verificar que ambas definiciones son equivalentes:

```
prop_strToInteger :: String -> Property
prop_strToInteger m =
     m /= [] ==> strToInteger1 m == strToInteger m
```

Que al ejecutar en consola:

```
ghci> quickCheck prop_strToInteger
+++ OK, passed 100 tests.
```

Para el código de la función inversa también tenemos dos posibilidades:

```
integerToStr1 :: Integer -> String
integerToStr1 0 = []
integerToStr1 n = integerToStr1 (div n 1000)
                  ++ [chr (fromIntegral (rem n 1000))]

integerToStr :: Integer -> String
integerToStr m | x == 0 = aux s
               | x == 1 = chr (read (take 1 s)) : aux (drop 1 s)
               | x == 2 = chr (read (take 2 s)) : aux (drop 2 s)
    where x     = rem (length (show m)) 3
          s     = show m
          aux n = map (chr . read) (chunksOf 3 n)
```

Nos quedaremos con la segunda opción ya que es más eficiente. Pero, al igual que antes, vamos a verificar que ambas funciones son equivalentes mediante la siguiente propiedad:

```
prop_integerToStr :: Integer -> Property
prop_integerToStr n =
     n>0 ==>  integerToStr1 n == integerToStr n
```

La comprobación es:

```
ghci> quickCheck prop_integerToStr
+++ OK, passed 100 tests.
```

Antes de poder usar estas funciones nos faltaría comprobar que realmente una es la inversa de la otra, y viceversa. Para ello definimos las dos siguientes propiedades:

```
prop_strToInteger_integerToStr :: String -> Property
prop_strToInteger_integerToStr m =
     m /= [] && 0 /= (ord . head) m ==>
       m == (integerToStr . strToInteger) m

prop_integerToStr_strToInteger :: Integer -> Property
prop_integerToStr_strToInteger n =
     n > 0 ==> n == (strToInteger . integerToStr) n
```

Que al ejecutarlas en consola:

```
ghci>  quickCheck prop_strToInteger_integerToStr
+++ OK, passed 100 tests.
ghci> quickCheck prop_integerToStr_strToInteger
+++ OK, passed 100 tests.
```

Y de esta forma, podemos comenzar a encriptar con un alto grado de confianza en que estas funciones están bien definidas.

2.3. Criptosistema RSA

2.3.1. Introducción

El **criptosistema RSA** es un criptosistema de clave pública, también conocidos como **criptosistemas asimétricos**. El algoritmo fue descrito por primera vez en 1977 por sus tres autores: Ron Rivest, Adi Shamir y Leonard Adleman; a los que debe su nombre.

Es el primer y más utilizado algoritmo de este tipo y sirve tanto para encriptar como para firmar digitalmente. Fue patentado por el Instituto Tecnológico de Massachusetts en 1983 quedando restringido su uso hasta el año 2000, que caducó dicha patente.

La seguridad del criptosistema radica en la dificultad que presenta la factorización de números grandes. Ya que el funcionamiento se basa en el producto, conocido, de dos números primos grandes elegidos al azar y mantenidos en secreto. Actualmente estos primos son del orden de 10^{200}, y se prevé que su tamaño crezca con el aumento de la capacidad de cálculo de los ordenadores.

2.3.2. Descripción matemática

En el sistema RSA no hay ningún dato compartido por todos los usuarios. Cada uno de ellos genera su clave de la siguiente forma:

1. Escogemos dos primos p y q.
2. Los multiplicamos entre sí: $N = p \times q$
3. Calculamos $\varphi(N) = (p-1) \times (q-1)$
4. Escogemos un número aleatorio E, primo con $\varphi(N)$
5. Hallamos d tal que se verifica: $E \times d = 1 \mod \varphi(N)$

El usuario que ha creado la clave hará públicos N y E, y ocultará el número d obtenido. De esta forma queda definida la clave pública como el par $K_1 = (N, E)$ y la clave privada como el par $K_2 = (N, d)$.

Cuando algún otro usuario quiera enviarnos un mensaje m sólo deberá coger nuestra clave pública (N, E) y realizar la siguiente operación (donde M es el mensaje ya encriptado) :

$$M = m^E \mod N$$

Huelga decir que hemos asociado al mensaje m un número entero que por abuso de notación lo llamaremos también m. Además, este número debe cumplir que $m < N$ para que este procedimento sea inyectivo. Si no se cumple bastará con dividir el mensaje en sub–mensajes más pequeños que cumplan dicha propiedad.

Una vez recibido M lo desencriptamos elevando M a d módulo N, quedando:

$$M^d = (m^E)^d = m^{E \times d} = m^{1+r \times \varphi(N)} = m \mod (N)$$

Me gustaría destacar que el proceso de encriptado y desencriptado son análogos, pero cambiando las claves de pública a privada. Esto es una propiedad muy útil a la hora de implementar dicho algoritmo.

2.3.3. Implementación

Análogamente a la sección anterior, en primer lugar implementaremos un generador de claves públicas y privadas.

El problema de generar primos aleatorios en la programación funcional no es para nada trivial. Afortunadamente estamos implementando en Haskell y nos podemoa ayudar de su inmenso catálogo de librerías. En este caso nos ha hecho falta acudir a la librería específica Codec.Crypto.RSA.Exceptions[1] de la que usaremos únicamente la función largeRandomPrime:

```
largeRandomPrime :: CryptoRandomGen g => g -> Int -> (Integer, g)
```

Esta función recibe un generador aleatorio g y un entero n y nos devuelve un par (g', p) donde g' es otro generador aleatorio y p es un número primo (cuyo tamaño está relacionado con n).

Como ya hemos explicado, el número E es un entero positivo primo con $\varphi(N)$. Por tanto, nos hace falta definir una función que se verifique si y sólo si dos números son primos entre sí:

```
coprimos :: Integral a => a -> a -> Bool
coprimos a b = gcd a b == 1
```

Por ejemplo,

```
ghci> coprimos 5 7
True
ghci> coprimos 21 77
False
```

Implementemos ahora la función `ejemploE` que nos devuelve una mónada que contiene al número E buscado:

[1] https://hackage.haskell.org/package/RSA-2.2.0/docs/Codec-Crypto-RSA-Exceptions.html

```
ejemploE :: Integer -> Gen Integer
ejemploE phiN =
    suchThat (choose (1,phiN)) (coprimos phiN)
```

Como trabajar con mónadas a veces puede resultar incómodo, implementamos la función `ejemploE'` que extrae la solución de la mónada:

```
ejemploE' :: Integer -> Integer
ejemploE' phiN =
    unsafePerformIO (generate (ejemploE phiN))
```

Que, por ejemplo, genera los siguientes números:

```
ghci> ejemploE' 30
13
ghci>ejemploE' 30
7
ghci>ejemploE' 30
17
```

Ya estamos en condiciones de implementar un generador de claves. Como las claves pública y privada no son independientes, nuestro generador nos devolverá un par con ambas. Igual que con la función anterior el generador nos devuelve una mónada, y necesitamos de una función auxiliar para extraer las claves de la mónada. Controlaremos el tamaño de los números con un entero (m) que daremos de entrada; así que, el código de las dos funciones es:

```
clavePubYPrivRSA :: Int -> IO ((Integer, Integer),(Integer, Integer))
clavePubYPrivRSA m = do
  g1 :: GenAutoReseed HashDRBG HashDRBG <- newGenIO
  let (p,g2) = largeRandomPrime g1 m
      (q,_)  = largeRandomPrime g2 m
      n = p*q
      phiN = (p-1)*(q-1)
      e = ejemploE' phiN
      d = invMod e phiN
  return ((n,e),(n,d))

clavePubYPrivRSA' :: Int -> ((Integer, Integer), (Integer, Integer))
clavePubYPrivRSA' m =
    unsafePerformIO (clavePubYPrivRSA m)
```

Veamos algunos ejemplos de claves generadas:

```
ghci> clavePubYPrivRSA' 3
((232042748932469,33057201511087),(232042748932469,51839958891703))
(0.03 secs, 29646792 bytes)
ghci> clavePubYPrivRSA' 3
((206168578675303,74107059957067),(206168578675303,135792506971123))
(0.03 secs, 35687160 bytes)
ghci> clavePubYPrivRSA' 20
((14008658197335418170608069592368193964154611229854785673582538992 1
  66456491896964714004485210616 69,10204095632317918070216429204566 43
  35281135517119706401928444766591679562359260656195152622042072 7),
 (14008658197335418170608069592368193964154611229854785673582538992 1
  66456491896964714004485210616 69,9397966354287029090737273364432636
  91904497950460395105870610836378521650034694164358897959125311))
```

Una vez generadas las claves podemos tratar de implementar la función encriptadora. Para ello necesitaremos de una función auxiliar que calcule las exponenciaciones modulares de una forma más eficiente. El código de dicha función es:

```
expQuickMod :: Integer -> Integer -> Integer -> Integer
expQuickMod x 1 n = x `mod` n
expQuickMod x e n | even e    = expQuickMod (x*x `mod` n) (div e 2) n
                  | otherwise = (x * expQuickMod x (e-1) n) `mod` n
```

Como estamos acostumbrados, podemos verificar con ayuda del QuickCheck que esta exponenciación rápida está bien definida. Para ello definimos en Haskell la siguiente propiedad:

```
prop_Exp :: Integer -> Integer -> Integer -> Property
prop_Exp x e n =
    e > 0 && n > 0 ==>
    expQuickMod x e n == x^e `mod` n
```

Y su comprobación es

```
ghci> quickCheck prop_Exp
+++ OK, passed 100 tests.
```

Por tanto podemos usar la función con cierta seguridad de que es correcta.

Una vez sentadas las bases, ya podemos implementar la función encriptadora que dado un mensaje m y una clave K_1 nos devuelve el mensaje encriptado. El código quedaría de la siguiente forma:

```
encriptaRSA ::  String -> (Integer,Integer) -> String
encriptaRSA m (n,e) = integerToStr (expQuickMod (strToInteger m) e n)
```

Mientras que el código de la función desencriptadora es:

```
desencriptaRSA :: String -> (Integer,Integer) -> String
desencriptaRSA = encriptaRSA
```

Veamos ahora un ejemplo de cómo funcionan:

```
ghci> let ((n,e),(m,d)) = clavesPubYPrivRSA' 100
ghci> ((n,e),(m,d))
 ((3076528004274436...46644911592466596224255 9,
   6917695950380385503...24297119472354022715),
  (30765280042...115924665962242559,
   107207592797466...36750757525519941007667)
ghci> encriptaRSA "Las matematicas puras son, en su forma, la poesia
 de las ideas logicas" (n,e)
" DC3 684 607 605 545 378 408 179 261 180 418Z 769 879
  557 580 323 678 244 713 764 618 291 522 718 898 862
  310 906 786 979 547 763 535 736 651 546 976 713 666
  346 185 560 416 841 769 247 364 337 998 559 436 340
  452 771 775 802 249 451 718 556 g J 343 905 788 663 697
  554 955 X 354 610 354 545 187 324 403 547 508 612 220
  V 973 611 849 584 862 J 328 719 911 530 461 347 616 741
  500 398 514 514 240 705 619 200 S O 217 DC4 700 862
  829 447 201 482 932 662 677 134 510 218 650 619 525
  525 542 887 165 647 967 418 977 ] 167 410 494 777 323
  203 705 140 273 722 843 454 590 402 590 487 356 323
  800 257 914 608 192 879 196 410 663 DC1 967"
ghci> desencriptaRSA it (n,e)
"Las matematicas puras son, en su forma, la poesia de las ideas logicas"
```

2.3.4. Corrección

Implementamos una propiedad para verificar la corrección del criptosistema para mensajes no vacíos, que no empiecen por el carácter `\NUL` y de longitud acotada. Como ya hemos comentado, además tenemos la necesidad de que el número entero correspondiente al mensaje m sea menor que el entero n (por el que hacemos módulo) para

garantizar inyectividad de la función encriptadora. Por tanto, la cota depende de n. También vamos a necesitar un entero x que definirá el tamaño de las claves y lo vamos a acotar por 30.

Teniendo todo lo anterior en cuenta la propiedad queda implementada de la siguiente forma:

```
prop_CorrecRSA :: Int -> String -> Property
prop_CorrecRSA x m =
    m /= [] && x>0 && x<30 && (ord (head m) > 0) &&
     (length m < div ((length . show) a) 3) ==>
      m == desencriptaRSA (encriptaRSA m (a,b)) (c,d)
    where ((a,b),(c,d)) = clavePubYPrivRSA' x
```

Que al ejecutar en consola:

```
ghci> quickCheck prop_CorrecRSA
+++ OK, passed 100 tests.
(8.80 secs, 9869952232 bytes)
```

Y por tanto podemos aumentar la confianza en que las funciones encriptadora y desencriptadora funcionan como deseamos.

2.4. Criptosistema de ElGamal

2.4.1. Introducción

El **criptosistema de ElGamal** es un criptosistema de clave pública o criptosistema asimétrico. El algoritmo fue descrito por *Taher Elgamal* en 1984, y se basa en el protocolo criptográfico *Diffie–Hellman*. Éste a su vez, fundamenta su seguridad en el famoso problema del logaritmo discreto.

Este criptosistema nos ofrece una alternativa al RSA para poder encriptar mensajes y hacer firmas digitales. Tiene una ventaja respecto al anterior y es que si repetimos el proceso de encriptado con un mismo mensaje obtenemos un mensaje cifrado distinto cada vez, este fenómeno se conoce como **encriptación aleatorizada** o probabilística.

Una desventaja muy importante que tiene es que el mensaje cifrado suele ser el doble de largo que el original. Esto nos daría una forma de averiguar la longitud del mensaje que queremos transmitir, pero solucionaremos esto implementando una pequeña variante del algoritmo de cifrado original.

2.4.2. Descripción matemática

En el cifrado de ElGamal todos los usuarios del sistema comparten cierta información, que llamaremos **entorno**:

1. Se escoge un primo p y el correspondiente cuerpo finito $\mathbb{F}_p$
2. Se elige otro primo q, lo más grande posible, de forma que $q \mid (p-1)$.
3. Se escoge una base $g \in \mathbb{F}_p^*$ con orden divisible por q. Equivalentemente:

$$g^{(p-1)/q} \neq 1 \mod p$$

Para que un usuario cree su clave pública (e) sólo debe escoger un entero n, con la única condición de que $1 < n < p-1$ y, posteriormente, realizar la siguiente operación:

$$e = g^n \mod p$$

Por otro lado, el usuario que quiera mandar el mensaje m encriptado debe:

1. Generar un clave efímera h (un entero cualquiera)
2. Calcular:

$$M = (M_1, M_2) = (g^h, m \cdot e^h)$$

Mientras que el usuario que recibe el mensaje encriptado $M = (M_1, M_2)$ lo descifra calculando:

$$\frac{M_2}{M_1^n} = \frac{m \cdot e^h}{g^{h \cdot n}} = m \mod p$$

Me gustaría destacar otra ventaja que tiene este criptosistema que es que las elecciones de parámetros deben de ser cuidadosas (p,q y g) las realiza el sistema y no el usuario. Éste sólo debe escoger números enteros (n ó h) que no afectan a la seguridad del cifrado.

2.4.3. Implementación

En primer lugar, y análogamente a como se ha descrito el cristosistema, trataremos de implementar una función que nos genere lo que denominamos *entorno*. Para necesitamos de otra antes que, a partir de p y q nos encuentre una base g en las condiciones ya descritas. En el código de esta función usaremos `expQuickMod`, que está descrita en la página 65, quedando de la siguiente forma:

```
baseG :: Integer -> Integer -> Integer
baseG p q = unsafePerformIO (generate (baseG' p q))
  where baseG' p q = suchThat
                       (choose (1,p-1))
                       (\x -> 1 /= expQuickMod x (div (p-1) q) p)
```

Por ejemplo:

```
ghci> baseG 7 2
6
ghci> baseG 7 2
3
ghci> baseG 7 2
5
```

Con la función anterior y la ayuda de `largeRandomPrime` descrita en la página 63 ya estamos en disposición de implementar una función que nos genere un *entorno* (de tamaño m) para el criptosistema.

```
generaEntorno :: Int -> (Integer, Integer, Integer)
generaEntorno m = unsafePerformIO (generaEntorno' m)
  where generaEntorno' m = do
          g1 :: GenAutoReseed HashDRBG HashDRBG <- newGenIO
          let (p,g2)  = largeRandomPrime g1 m
              (q,exp) = last (primeFactors TrialDivision (p-1))
```

```
            g       = baseG p q
        return (p,q,g)
```

Me gustaría destacar que un *entorno*, tal y como lo describimos en la sección anterior, queda unívocamente determinado por la terna (p, q, g). Unos ejemplos de entornos generados con esta función son:

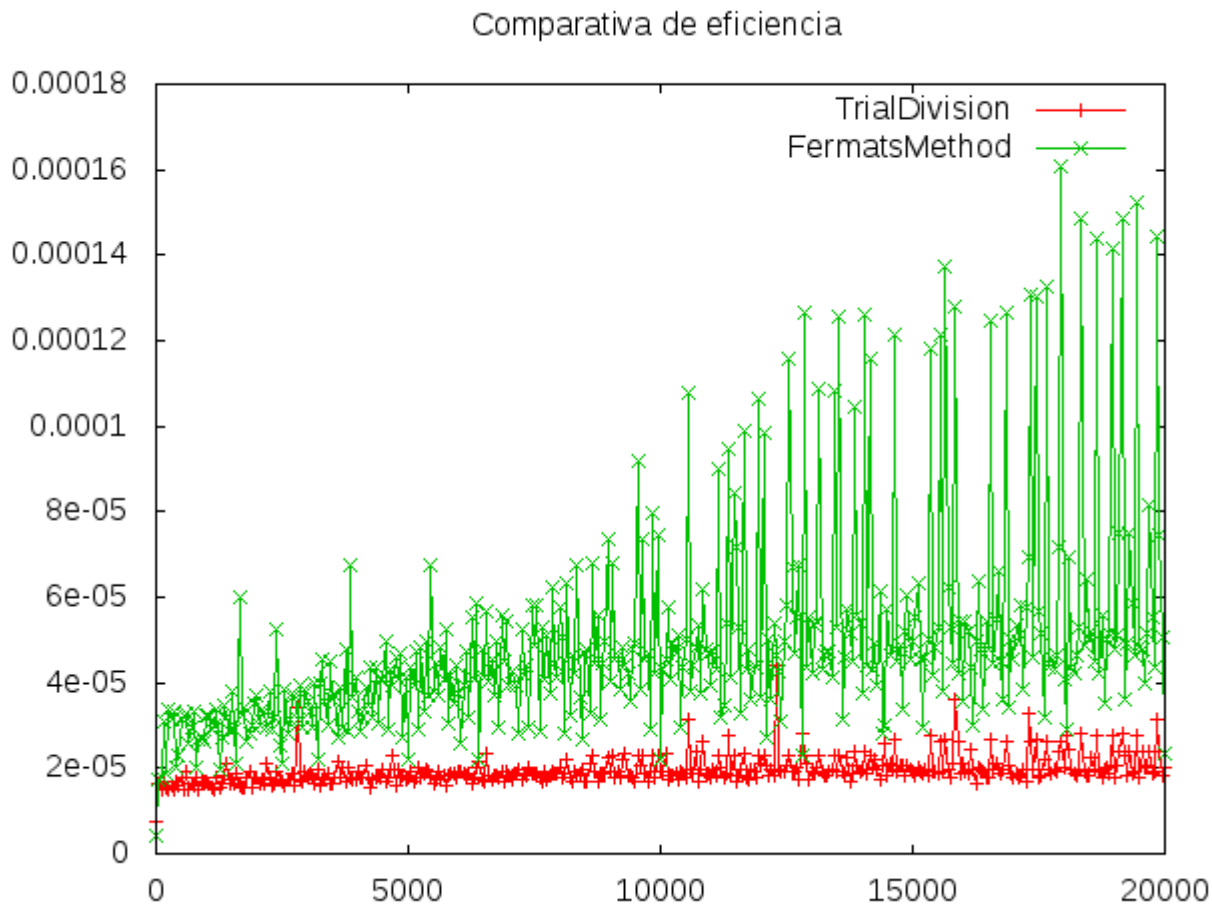
```
ghci> generaEntorno 2
(54151,19,38377)
ghci> generaEntorno 2
(51287,25643,23532)
ghci> generaEntorno 6
(233563020843869,58390755210967,82779205951658)
```

Observemos que el *tamaño* no lo hemos definido formalmente ya que es una noción intuitiva. A partir del tamaño $m = 6$ esta función se enlentece mucho debido al ineficiente cálculo de q. Para tratar de mejorarlo nos hemos ayudado de la extensa librería factory[2] que tiene implementados dos algoritmos para `primeFactors`: `TrialDivision` y `FermatsMethod`. Así que hacemos una pequeña comparativa:

```
ghci> primeFactors FermatsMethod 2351502222
[(2,1),(3,1),(163,1),(2404399,1)]
(2.07 secs, 2599323352 bytes)
ghci> primeFactors TrialDivision 2351502222
[(2,1),(3,1),(163,1),(2404399,1)]
(0.01 secs, 4672408 bytes)
ghci> primeFactors TrialDivision 23515022352
[(2,4),(3,1),(489896299,1)]
(0.02 secs, 11952264 bytes)
ghci> primeFactors FermatsMethod 23515022352
... (no acaba)
```

A la vista de este resultado tan desalentador, vamos a asegurarnos de que lo que nos ha salido no es casualidad. Para ello implementamos un comparador de eficiencia que mide el tiempo de ejecución de `primeFactors` usando cada uno de los dos algoritmos en números entre el 0 y el 20000. De esta forma, al dibujar estos tiempos de ejecución obtenemos la siguiente gráfica:

[2] https://hackage.haskell.org/package/factory



Es fácil ver a partir de la gráfica que la implementación del algoritmo de Fermat es mucho más lenta que la de fuerza bruta. Por tanto, nos conformaremos con la implementación que ya tenemos y dejaremos como ejercicio la implementación de un algoritmo más eficiente para el cálculo del mayor factor primo de un número dado (complejidad exponencial).

Una vez descrito el entorno, con más o menos eficiencia, el siguiente paso es implementar una función que nos genere tanto la clave pública como la privada. Por comodidad integraremos en la noción de clave los datos del entorno que nos sean necesarios para los cálculos posteriores. De esta forma, el código de dicha función es:

```
clavePubyPrivElGamal :: (Integer, t, Integer) ->
                    ((Integer, Integer, Integer),(Integer,Integer))
clavePubyPrivElGamal x = unsafePerformIO (clavePubyPrivElGamal' x)
  where clavePubyPrivElGamal' (p,q,g) = do
            let n = unsafePerformIO (generate (choose (1,p-1)))
            return ((p,g,expQuickMod g n p),(p,n))
```

Veamos unos ejemplos de cómo actúa esta función:

```
ghci> clavePubyPrivElGamal (13061663,6530831,4826906)
((13061663,4826906,5648996),(13061663,697850))
ghci> clavePubyPrivElGamal (13061663,6530831,4826906)
((13061663,4826906,691985),(13061663,3930900))
clavePubyPrivElGamal (909209565115226705262277,17530839358873691,
80323309362668445537568 1)
((909209565115226705262277,80323309362668445537568 1,
  393347751589997786171587),(909209565115226705262277,55670718812240120356451))
```

De esta forma ya hemos completado todos los preparativos para poder implementar la función encriptadora. Como ya comentamos en la página 68 escogeremos una clave efímera h que, por simplicidad, estará entre 1 y p. Además nos ayudaremos de las funciones `pack`, `unpack` y `chunksOf` de la librería específica text[3]. El código de la función que encripta usando el sistema de ElGamal es:

```
encriptaElGamal m (a,b,c) = foldr
                   (\z y-> encriptaElGamal' (unpack z) (a,b,c) : y) []
                           (chunksOf (div ((length . show) a) 3) (pack m))
 where encriptaElGamal' x y = unsafePerformIO  (encriptaElGamal'' x y)
       where encriptaElGamal'' mensaje (p,g,e) = do
              let h = unsafePerformIO (generate (choose (1,p)))
              return (integerToStr (expQuickMod g h p),
                      integerToStr (m' * expQuickMod e h p `mod` p))
              where m' = strToInteger mensaje
```

Mientras que el código de la función desencriptadora es:

```
desencriptaElGamal xs (a,b) =
 foldr (\m y -> (desencriptaElGamal' m (a,b) ++ y)) [] xs
 where desencriptaElGamal' (mensaje1,mensaje2) (p,n) =
         integerToStr (expQuickMod m1 (p-1-n) p * m2 `mod` p)
         where (m1,m2) = (strToInteger mensaje1, strToInteger mensaje2)
```

A continuación veremos unos ejemplos de cómo actúan las funciones encriptadora y desencriptadora:

```
ghci> encriptaElGamal "Jacobi" (909209565115226705262277,
80323309362668445537568 1,393347751589997786171587)
```

[3] https://hackage.haskell.org/package/text

```
[("236 788 921 543 288 837 512 410","400 807 621 190 360 396 294 392")]
ghci> desencriptaElGamal it (90920956511522670526277,
5567071881224012035645l)
"Jacobi"
ghci> encriptaElGamal "La matematica es la ciencia de lo que es claro
 por si mismo." (90920956511522670526277,80323309362668445537568l,
 393347751589997786171587)
[("211 C SI 353 428 217 689 635","146 ? 596 479 832 139 c 922"),
(" 398 ESC 496 455 ) 686 790 239","750 325 870 188 256 580 } 814"),
("682 298 443 544 482 c 362 761","t 903 & 0 v 348 799 825 514"),
("860 233 461 830 991 661 GS 814","598 " 195 & 8 741 272 145 677"),
("508 591 g 473 Q 249 710 ETX"," 227 803 567 v y 278 860 785"),
(" 427 v 950 901 281 992 786 429","@ 668 416 150 671 183 330 352"),
(" " 901 606 185 H 652 181 622","u 893 c 580 383 809 420 SYN"),
("429 888 824 390 559 747 302 366","135 329 g 629 610 204 765 137")]
ghci> desencriptaElGamal it (90920956511522670526277,
5567071881224012035645l)
"La matematica es la ciencia de lo que es claro por si mismo."
```

2.4.4. Corrección

Implementamos una propiedad para verificar la corrección del criptosistema para mensajes no vacíos y que no empiecen por el carácter `\NUL`.

Debido a la ineficiencia de la función `generaEntorno` vamos a definir dos propiedades: una que compruebe la corrección del criptosistema para un entorno prefijado con anterioridad de tamaño 11; y otra que use entornos variables de tamaño dado (pequeño).

Teniendo todo lo anterior en cuenta la primera propiedad queda implementada de la siguiente forma:

```
prop_CorrecElGamal1 :: String -> Property
prop_CorrecElGamal1 m =
  m /= [] && all (\ x -> ord x > 0) m ==>
    m == desencriptaElGamal (encriptaElGamal m clavPub) clavPriv
      where (clavPub,clavPriv) =
              clavePubyPrivElGamal (90920956511522670526277,
                  17530839358873691,80323309362668445537568l)
```

Mientras que el código de la segunda propiedad es:

```
prop_CorrecElGamal2 :: Int -> String -> Property
prop_CorrecElGamal2 n m =
       m /= [] && all (\ x -> ord x > 0) m  ==>
       m == desencriptaElGamal (encriptaElGamal m clavPub) clavPriv
         where (clavPub,clavPriv) = clavePubyPrivElGamal (generaEntorno n)
```

Que al ejecutar ambas en consola:

```
ghci> quickCheck prop_CorrecElGamal1
+++ OK, passed 100 tests.
(0.34 secs, 173203352 bytes)
ghci>  quickCheck (prop_CorrecElGamal2 5)
+++ OK, passed 100 tests.
(2.63 secs, 3410869656 bytes)
```

2.5. Criptosistema de Rabin

2.5.1. Introducción

El **criptosistema de Rabin** es un criptosistema de clave pública, también conocidos como **criptosistemas asimétricos**. El algoritmo de este sistema criptográfico fue publicado en enero de 1979 por el investigador de origen israelí Michael O. Rabin; quien, junto con Dana Scott, había ganado el premio Turing por su trabajo sobre autómatas no deterministas un par de años antes.

La seguridad del criptosistema se basa en los problemas de la factorización y en el de las raíces cuadradas modulares; y es que, en realidad, ambos problemas están muy relacionados. De hecho, en algunos círculos se conoce a este criptosistema como una variante del RSA.

La ventaja del criptosistema de Rabin frente al RSA es que se ha podido demostrar que la complejidad del problema en el que se basa es la misma que la de la factorización de enteros, cosa que se desconoce si es cierto en el caso del RSA simple.

El inconveniente que tiene es que cada salida de la función de Rabin puede ser generado por 4 posibles entradas, y si cada salida es un texto cifrado se requiere un tiempo extra en el descifrado para identificar cual de las 4 posibles entradas era el correcto texto en claro. Nosotros resolveremos este detalle dejando al usuario identificar cuál es el mensaje correcto.

2.5.2. Descripción matemática

En el sistema Rabin, al igual que en el RSA, no hay ningún dato compartido por todos los usuarios. Cada uno de ellos genera su clave de la siguiente forma:

1. Escogemos dos primos p y q tales que $p = q = 3 \mod 4$.
2. Calculamos: $n = p \times q$.
3. Escogemos un entero e tal que $0 < e < n$.
4. La clave pública es el par (n, e). Mientras que la clave privada contiene toda la información: (p, q, n, e).

Si alguien quiere enviar un mensaje a otro usuario deberá calcular M de la siguiente forma:

$$M = m \times m \mod n$$

Mientras que el que recibe el mensaje deberá hallar la raíz cuadrada:

$$\sqrt{M} = \sqrt{m^2} = m \mod n$$

Si analizamos detenidamente el proceso anterior nos damos cuenta de que da demasiada información sobre m si $m \ll n$ pues bastaría con calcular la raíz cuadrada usual (no modular). Afortunadamente hay una variante de encriptación que hace algo más difícil el ataque al criptosistema. Simplemente sumaremos el número aleatorio e a nuestro mensaje m. Quedando el cálculo del mensaje encriptado de la siguiente forma:

$$M = m \times (m + e) \mod n$$

Y la operación de desencriptado es:

$$\sqrt{\frac{e^2}{4} + M} - \frac{e}{2} = \sqrt{\frac{(2m+e)^2}{4}} - \frac{e}{2} = \frac{(2m+e)}{2} - \frac{e}{2} = m \mod n$$

Observemos que ya introdujimos al número e en las claves pues sabíamos que iba a ser necesario para la variante. Si no quisiésemos usar esta segunda forma de encriptación la clave pública sería n y la privada sería (p, q, n).

2.5.3. Implementación

En primer lugar vamos a tratar de programar unas funciones que nos generen las claves pública y privada. Para ello, necesitaremos de una función que, dado un número n, nos escoja otro al azar en el intervalo $(1, n)$. El código de esta función es:

```
choose' :: (Integral a, Random a) => a -> a
choose' n = unsafePerformIO (aux n)
    where aux n = getStdRandom (randomR (1,div n 2))
```

Veamos a continuación unos ejemplos de su funcionamiento:

```
ghci> choose' 10
1
ghci> choose' 10
5
ghci> choose' 1000
474
```

La siguiente función necesaria es una que compruebe si un primo dado es congruente con 3 módulo 4 (propiedad necesaria para aplicar el algoritmo de Shanks). Esta función, en caso de que el primo dado no cumpla la propiedad, nos generará automática otro primo y repetirá el proceso. La implementamos así:

```
prim3Mod4 :: CryptoRandomGen t => Int -> (Integer, t) -> Integer
prim3Mod4 m (p,g) | p `mod` 4 == 3 = p
```

```
                  | otherwise      = prim3Mod4 m x
    where x = largeRandomPrime g m
```

Gracias a estas dos funciones y a `largeRandomPrime` comentada en la página 63 ya estamos en disposición de generar el par de claves. El código de la función que, especificado un tamaño, nos devuelve un par de claves del criptosistema de Rabin es:

```
clavesPubyPrivRabin :: Int ->
             ((Integer,Integer),(Integer,Integer,Integer,Integer))
clavesPubyPrivRabin m = unsafePerformIO (aux m)
  where aux m = do
          g1 :: GenAutoReseed HashDRBG HashDRBG <- newGenIO
          g2 :: GenAutoReseed HashDRBG HashDRBG <- newGenIO
          let (p0,g3) = largeRandomPrime g1 m
              p       = prim3Mod4 m (p0,g3)
              (q0,g4) = largeRandomPrime g2 m
              q       = prim3Mod4 m (q0,g4)
              n       = p*q
              e       = 2 * choose' n
          return ((n,e),(p,q,n,e))
```

Que por ejemplo genera:

```
ghci> clavesPubyPrivRabin 3
((163942103951057,48435145827478),
 (12813023,12794959,163942103951057,48435145827478))
ghci> clavesPubyPrivRabin 5
((923367885121449262832809,741916019821220406385592),
 (967554033791,954332112599,923367885121449262832809,
  741916019821220406385592))
```

Una vez implementado el generador de claves, el siguiente paso es dar el código de las funciones encriptadora y desencriptadora. Aunque para ello necesitamos una función que nos devuelva las lista de posibles raíces cuadradas modulares. Esta función usará el algoritmo de Shanks para calcularlas (usualmente son 4). Hemos acudido a la librería arithmoi[4] de la que usaremos únicamente la función `sqrtModFList` que hace lo que queremos:

```
sqrtModFList :: Integer -> [(Integer, Int)] -> [Integer]
```

[4] http://hackage.haskell.org/package/arithmoi.html

Veamos cómo calcula las raíces cuadradas de 16 módulo 153 (es necesario darle la descomposición en producto de primos $153 = 17^1 \times 3^2$)

```
ghci> sqrtModFList 16 [(17,1),(3,2)]
[149,13,140,4]
ghci> 149*149 `mod` 153
16
ghci> 13*13 `mod` 153
16
ghci> 140*140 `mod` 153
16
ghci> 4*4 `mod` 153
16
```

Tal y como se comentó en la descripción matemática hay una primera posibilidad de encriptar (sin usar el número e) y una variante más segura (que se ayuda del número aleatorio e). Comenzaremos implementando la primera posibilidad, quedando el código de la función encriptadora de la siguiente forma:

```
encriptaRabin1 :: (Integer, t) -> String -> String
encriptaRabin1 (n,e) ms = integerToStr ((m*m) `mod` n)
    where m = strToInteger ms
```

Mientras que el código de la función desencriptadora es:

```
desencriptaRabin1 :: (Integer, Integer, Integer, t) -> String -> [String]
desencriptaRabin1 (p,q,n,e) ms =
  foldr (\x y -> integerToStr (x `mod` n) : y) [] raices2
      where m       = strToInteger ms
            raices2 = sqrtModFList m [(p,1),(q,1)]
```

Veamos ahora unos ejemplos de cómo funcionan estas funciones:

```
ghci> encriptaRabin1 (102782853580532303143568163967683210441278977168l,
10139957542759686375915305787740164218414063937O2) "Rabin"
"ACK 739 933 517 280 &7 952 608 p d"
ghci> desencriptaRabin1 (967874814038127979355507,
106194367380741995933788 3,102782853580532303143568163967683210441278977168l,
10139957542759686375915305787740164218414063937O2) it
["Rabin","266 613 * 446 824 922 456 131 230 195 735 254 490 178 875 450",
 "761 215 493 358 498 l 979 550 409 481 ‘ 849 922 610 896 231",
 "SOH ESC 828 535 805 323 US 435 681 639 676 832 SYN 315 691 666 571"]
```

Ahora implementaremos la variante, que es más segura. El código de la función encriptadora es:

```
encriptaRabin :: (Integer, Integer) -> String -> String
encriptaRabin (n,e) ms = integerToStr ((m*(m+e)) `mod` n)
    where m = strToInteger ms
```

Y la función desencriptadora la implementamos de la siguiente manera:

```
desencriptaRabin
  :: (Integer, Integer, Integer, Integer) -> String -> [String]
desencriptaRabin (p,q,n,e) ms =
  foldr (\x y -> integerToStr ((x-e_2) `mod` n) : y) [] raices2
  where e2_4    = div (e^2) 4
        e_2     = div e 2
        m       = strToInteger ms
        raices2 = sqrtModFList ((m + e2_4) `mod` n) [(p,1),(q,1)]
```

Ahora veamos un ejemplo de cómo actúan estas funciones:

```
ghci> encriptaRabin (3534195011492...81078753,123443873...80531696)
"Los encantos de esta ciencia sublime, las matematicas, solo se le revelan
 a aquellos que tienen el valor de profundizar en ella."
" " 829 797 317 963 625 985 474 519 384 617 237 504 157 998 496 983 370
 169 142 750 897 448 331 319 324 984 425 248 864 267 875 422 481 421 &7
 409 915 939 958 880 976 331 737 377 784 615 269 ETX b 250 334 980 269
 179 582 514 573 508 550 962 349 237 d 781 s 966 901 588 n 685 431 286 f
 L NUL  294 D 818 213 381 547 S 511 816 205 818 881 726 453 296 442 543
 278 486 332 781 DEL 337 828 658 958 945 NUL 890 A p 259 ETX 860 288 950
 674 # 789 910 &8 650 151 362 443 411 529 720 164 t 196 311 + 966 890 360
 719 579 EM { 367 784 891 246 & 261 155 ' 714 990 854 520 727 782 829
 937 693 196 388 275 883 413 192 603"
ghci> desencriptaRabin (5355545...22301543,6599131107...68164471,
3534195011492...81078753,1234438735...0531696) it
["RS ~ 325 887 392 567 &0 641 ~ 340 667 s SOH ... x 650 549 763 391 722 457",
 "Los encantos de esta ciencia sublime, las matematicas, solo se le revelan
  a aquellos que tienen el valor de profundizar en ella.",
 "SYN 997 562 763 865 w ... DC4 983 371  450 v",
 "FS  213 186 991 ` 397 m ... > 364 G 618 748 b . FSR 623"]
```

Y como vemos es la segunda desencriptación la correcta, ya que es la única que tiene sentido.

2.5.4. Corrección

Implementamos una propiedad para verificar la corrección del criptosistema para mensajes no vacíos, que no empiecen por el carácter `\NUL` y de longitud acotada. Como ya hemos comentado, además tenemos la necesidad de que el número entero correspondiente al mensaje m sea menor que el entero n (por el que hacemos módulo) para garantizar la inyectividad de la función encriptadora. Por tanto, la cota depende de n. También vamos a necesitar un entero x que definirá el tamaño de las claves y lo vamos a acotar por 30.

Teniendo todo lo anterior en cuenta la propiedad queda implementada de la siguiente forma:

```
prop_CorrecRabin :: Int -> String -> Property
prop_CorrecRabin x m =
    m /= [] && 0 < x && x < 30 && (ord (head m) > 0) &&
     (length m < div ((length . show) a) 3) ==>
      elem m (desencriptaRabin (c,d,e,f) (encriptaRabin (a,b) m))
    where ((a,b),(c,d,e,f)) = clavesPubyPrivRabin x
```

Que al ejecutar en consola:

```
ghci> quickCheck prop_CorrecRabin
+++ OK, passed 100 tests.
(13.76 secs, 15876675976 bytes)
```

Apéndice A

Códigos

En este apéndice se incluye los códigos de los programas desarrollados en el trabajo.

A.1. Asociacion1

```
int2char :: Int -> Char
int2char n = chr (n + 65) -- ord('A') = 65

char2int :: Char -> Int
char2int c = ord c - 65

int2str :: [Int] -> String
int2str = map int2char

str2int :: String -> [Int]
str2int = map char2int

abecedario :: String
abecedario = ['A'..'Z']

enAbecedario :: String -> Bool
enAbecedario [] = False
enAbecedario xs = all (`elem` abecedario) xs

mcdExt :: Integral a => a -> a -> (a,a,a)
mcdExt a 0 = (1, 0, a)
mcdExt a b = (t, s - q * t, g)
  where (q, r)    = a `quotRem` b
```

```
        (s, t, g) = mcdExt b r

invMod :: Integral a => a -> a -> a
invMod a m | x < 0     = x + m
           | otherwise = x
    where (x, _, _) = mcdExt a m

invmod1 :: Integer -> Integer -> Integer
invmod1 a n
  | 1 == gcd a n = head [m | m <- [1..n-1], m*a `mod` n == 1]
  | otherwise    = error "No existe"

prop1_invMod :: Positive Integer -> Integer -> Property
prop1_invMod (Positive a) n =
  n > 1 && gcd a n == 1 ==> invMod a n == invmod1 a n

prop2_invMod1 :: Positive Integer -> Integer -> Property
prop2_invMod1 (Positive a) n =
  n > 1 && gcd a n == 1 ==>
    0 <= x && x < n &&
    x * a `mod` n == 1
  where x = invMod a n
```

A.2. Asociacion2

```
bin2int =  foldr (\x y -> x + 2*y) 0

bin2intR :: [Int] -> Int
bin2intR [] = 0
bin2intR (x:xs) = x + 2 * bin2intR xs

bin2intC :: [Int] -> Int
bin2intC xs = sum [x*2^n | (x,n) <- zip xs [(0::Int)..]]

int2bin :: Int -> [Int]
int2bin n | n < 2     = [n]
          | otherwise = n `mod` 2 : int2bin (n `div` 2)

prop_int2bin :: Int -> Bool
prop_int2bin x =
```

```
  bin2int (int2bin y) == y
  where y = abs x

creaOcteto :: [Int] -> [Int]
creaOcteto bs =  take 8 (bs ++ repeat 0)

creaOcteto' :: [Int] -> [Int]
creaOcteto' bs =  take 8 (bs ++ replicate 8 0)

codifica :: String -> [Int]
codifica =  concatMap (creaOcteto . int2bin . ord)

separaOctetos :: [Int] -> [[Int]]
separaOctetos [] = []
separaOctetos bs =
  take 8 bs : separaOctetos (drop 8 bs)

descodifica :: [Int] -> String
descodifica =  map (chr . bin2int) . separaOctetos
```

A.3. Asociacion3

```
xor :: [Int] -> [Int] -> [Int]
xor xs ys = [xor' x y | (x,y) <- zip xs ys]
  where xor' x y = bool2int (x /= y)

int2bin4' :: Integral a => a -> [a]
int2bin4' n | n < 2     = [n]
            | otherwise = int2bin4' (n `div` 2) ++ [n `mod` 2]

largo4 :: Num a => [a] -> [a]
largo4 xs | length xs >= 4 = xs
          | otherwise      = largo4 (0:xs)

int2bin4 :: Integral a => a -> [a]
int2bin4 = largo4 . int2bin4'

bin2int4 :: [Int] -> Int
bin2int4 xs = foldr (\x y -> x + 2*y) 0 (reverse xs)
```

```
prop_int2bin4 :: Int -> Bool
prop_int2bin4 x =
  bin2int4 (int2bin4 y) == y
  where y = abs x
```

A.4. Criptoanalisis

```
aparicion :: Char -> String -> Int
aparicion a xs = length (filter (==a) xs)

frecuencias :: String -> [(Int,Char)]
frecuencias m = [(aparicion a m,a) | a <- abecedario]

ordena2 :: [(Int,Char)] -> String
ordena2 xs = map snd (sortBy (flip compare) xs)

frecMayores :: String -> String
frecMayores m = [y | (x,y) <- frecuencias m, x == frecMayor]
  where frecMayor = maximum [aparicion a m | a <- abecedario]

aparicion2 :: String -> String -> Int
aparicion2 _ [] = 0
aparicion2 xs (y:ys)
  | xs `isPrefixOf` (y : ys) = 1 + aparicion2 xs ys
  | otherwise                = aparicion2 xs ys
```

A.5. Generadores

```
genLetra :: Gen Char
genLetra = elements abecedario

mensaje :: Gen String
mensaje = listOf genLetra
```

A.6. Cesar

```
import Test.QuickCheck
```

```
cesarF :: Int -> String -> String
cesarF k m = int2str [(n+k) `mod` 26 | n <- str2int m]

cesarG :: Int -> String -> String
cesarG k = cesarF (-k)

prop_CorreccionDeCesar :: Int -> Property
prop_CorreccionDeCesar k =
  k >= 0 ==>
    forAll mensaje (\m -> m == cesarG k (cesarF k m))

desencriptaCesarBruta :: String -> [String]
desencriptaCesarBruta m = [cesarG k m | k <- [0..25]]

desencriptaCesarFino :: String -> [String]
desencriptaCesarFino m =
  [cesarG (char2int k - 4) m | k <- frecMayores m]
```

A.7. Afin

```
import Test.QuickCheck

afinF :: (Int,Int) -> String -> String
afinF (k1,k2) m = int2str [((n*k1)+k2) `mod` 26 | n <- str2int m]

afinG :: (Int,Int) -> String -> String
afinG (k1,k2) m' =
  int2str [((n-k2) * invMod k1 26) `mod` 26 | n <- str2int m']

primocon26 :: Int -> Bool
primocon26 n = gcd n 26 == 1

mensajeYclaveOK :: (String,(Int,Int)) -> Bool
mensajeYclaveOK (m,(k1,_)) = enAbecedario m && primocon26 k1

mensajeYclaveA :: Gen (String,(Int,Int))
mensajeYclaveA =
  do m <- mensaje
     k1 <- suchThat (choose (0,26)) primocon26
     k2 <- choose (0,26)
```

```
     return (m,(k1,k2))

prop_CorreccionAfin :: Property
prop_CorreccionAfin =
  forAll mensajeYclaveA (\(m,k)-> m == afinG k (afinF k m))

letrasFrecuentes :: String -> String
letrasFrecuentes m = take 2 (ordena2 (frecuencias m))

despejak1 :: Int -> Int -> Int
despejak1 x y =
  invMod (head [k1 | k1 <- [1..25],
                     k1 * (x - y) `mod` 26 == 4,
                     gcd k1 26 == 1])
         26

posibleClave :: String -> (Int,Int)
posibleClave m = (despejak1 x y,y)
  where [x,y] = str2int (letrasFrecuentes m)

desencriptaAfin :: String -> String
desencriptaAfin m = afinG (posibleClave m) m
```

A.8. Vigenere

```
repite :: String -> String
repite = concat . repeat

vigenereF :: String -> String -> String
vigenereF k m = map (\x -> int2char (x `mod` 26)) (zipWith (+) xs ys)
  where xs = str2int m
        ys = str2int (repite k)

vigenereG :: String -> String -> String
vigenereG k m' = map (\x -> int2char (x `mod` 26)) (zipWith (-) xs ys)
  where xs = str2int m'
        ys = str2int (repite k)

mensajeOclaveV :: Gen String
```

```
mensajeOclaveV = listOf1 genLetra

prop_CorreccionVigenere :: Property
prop_CorreccionVigenere =
  forAll mensajeOclaveV
         (\k -> forAll mensajeOclaveV
                       (\m -> m == vigenereG k (vigenereF k m)))
```

A.9. Hill

```
import Test.QuickCheck

divModular :: Int -> Int -> Int -> Int
divModular a b n = head [m | m <- [0..n-1], m*b `mod` n == a]

invertible :: Matrix Int -> Bool
invertible a = 1 == gcd x 26
  where x = abs (detLaplace a)

matrizAdj :: Matrix Int -> Matrix Int
matrizAdj a =
  matrix (nrows a) (ncols a) $
  \(i,j)-> (((-1)^(i+j)) * detLaplace (minorMatrix i j a))

matrizAdjTrasp :: Matrix Int -> Matrix Int
matrizAdjTrasp = transpose . matrizAdj

matrizInv :: Matrix Int -> Matrix Int
matrizInv a
  | invertible a = matrizInvAux (matrizAdjTrasp a) (nrows a)
  | otherwise    = error "Matriz no valida"
  where
    matrizInvAux b 0 = b
    matrizInvAux b n =
        matrizInvAux (mapRow (\_ x -> ((x * t) `mod` 26)) n b) (n-1)
    t = invMod (detLaplace a `mod`26) 26

hillF :: Matrix Int -> String -> String
hillF xs n | invertible xs = aux xs n
           | otherwise     = error "Matriz no valida"
```

```
  where aux _ [] = []
        aux ks m | nrows ks <= length m =
                      int2str
                       (toList
                        (mapCol
                         (\_ x -> x `mod` 26)
                         1
                         (multStd ks (transpose (fromLists
                                  [str2int (take (nrows ks) m)]))
                                  ))) ++
                      aux ks (drop (nrows ks) m)
                 | otherwise = m

hillG :: Matrix Int -> String -> String
hillG ks = hillF (matrizInv ks)

matriz :: Gen (Matrix Int)
matriz = do
  n <- choose (2,6)
  xs <- suchThat (vector (n*n)) (esInvertible n)
  return (fromList n n xs)
  where esInvertible n xs = invertible (fromList n n xs)

mensajeYclaveHill :: Gen (String,Matrix Int)
mensajeYclaveHill = do
  m <- mensaje
  k <- matriz
  return (m,k)

prop_CorreccionHill :: Property
prop_CorreccionHill =
  forAll mensajeYclaveHill
         (\(m,k)-> m == hillG k (hillF k m))
```

A.10. Permutaciones

```
ordena :: Ord a => [(a,b)] -> [b]
ordena [] = []
ordena ((n,x):xs) = ordena anteriores ++ x : ordena posteriores
  where anteriores  = [(m,y) | (m,y) <- xs, m <= n]
```

```
        posteriores = [(k,z) | (k,z) <- xs, k > n]

permuta :: [Int] -> [a] -> [a]
permuta ns xs = ordena (zip ns xs)

permutaInv :: [Int] -> [Int]
permutaInv ns = permuta ns [1..]

permutacionF :: [Int] -> String -> String
permutacionF _  [] = []
permutacionF ks m
  | length ks <= length m =
      permuta ks (take (length ks) m) ++
      permutacionF ks (drop (length ks) m)
  | otherwise = m

permutacionG :: [Int] -> String -> String
permutacionG ks = permutacionF (permutaInv ks)

permutacion :: Gen [Int]
permutacion = do
  n <- choose (2,6)
  shuffle [1..n]

mensajeYclavePerm :: Gen (String,[Int])
mensajeYclavePerm = do
  m <- mensaje
  k <- permutacion
  return (m,k)

prop_CorreccionPerm :: Property
prop_CorreccionPerm =
  forAll mensajeYclavePerm
         (\(m,k)-> m == permutacionG k (permutacionF k m))
```

A.11. DES

```
import Test.QuickCheck

ejemploM :: String
```

```
ejemploM = "ABCEDFGH"

ejM :: [Int]
ejM = [1,1,0,0,1,1,0,0,0,0,0,0,0,0,0,0,1,1,0,0,1,1,0,0,1,1,1,1,1,1,1,1,1,1,1,1,
       0,0,0,0,1,0,1,0,1,0,1,0,1,1,1,1,0,0,0,0,1,0,1,0,1,0,1,0]

ejMensaje :: [Int]
ejMensaje = [0,0,0,0, 0,0,0,1, 0,0,1,0, 0,0,1,1, 0,1,0,0, 0,1,0,1, 0,1,1,0,
             0,1,1,1, 1,0,0,0, 1,0,0,1, 1,0,1,0, 1,0,1,1, 1,1,0,0, 1,1,0,1,
             1,1,1,0, 1,1,1,1]

mensaje_ej2 :: [Int]
mensaje_ej2 = [0,0,0,0,0,0,0,1,0,0,1,0,0,0,1,1,0,1,0,0,0,1,0,1,0,1,1,0,0,1,1,1,
               1,0,0,0,1,0,0,1,1,0,1,0,1,0,1,1,1,1,1,0,0,1,1,0,1,1,1,1,1,0,1,1,1,1]

clave_ej2 :: [Int]
clave_ej2 = [0,0,0,1,0,0,1,1,0,0,1,1,0,1,0,0,0,1,0,1,0,1,1,1,0,1,1,1,1,0,0,1,1,
             0,0,1,1,0,1,1,1,1,0,1,1,1,1,1,0,0,1,1,0,1,1,1,1,1,1,1,1,1,0,0,0,1]

ejemploK :: [Int]
ejemploK = [0,0,0,1,0,0,1,1,0,0,1,1,0,1,0,0,0,1,0,1,0,1,1,1,0,1,1,1,1,0,0,1,1,0,
            0,1,1,0,1,1,1,1,0,1,1,1,1,0,0,1,1,0,1,1,1,1,1,1,1,1,1,0,0,0,1]
ejemplok :: [Int]
ejemplok = [0,0,0,1,1,0,1,1,0,0,0,0,0,0,1,0,1,1,1,0,1,1,1,1,1,1,1,1,1,1,0,
            0,0,1,1,1,0,0,0,0,0,1,1,1,0,0,1,0]

ejemploL :: [Int]
ejemploL = [1,0,0,0,0,0,1,0,0,1,0,0,0,0,1,0,1,1,0,0,0,0,1,0,1,0,1,0,0,0,1,0]

ejemploR :: [Int]
ejemploR = [0,0,1,0,0,0,1,0,0,1,1,0,0,0,1,0,1,1,1,0,0,0,1,0,0,0,0,1,0,0,1,0]

mensaje_cifrado_ej2 :: [Int]
mensaje_cifrado_ej2 =
  [1,0,0,0,0,1,0,1,1,1,1,0,1,0,0,0,0,0,0,1,0,0,1,1,0,1,0,1,
   0,1,0,0,0,0,0,0,1,1,1,1,0,0,0,0,1,0,1,0,1,0,1,1,0,1,0,0,0,0,0,0,0,1,0,1]

ejemploencriptado :: [Int]
ejemploencriptado =
  [1,0,0,0,0,1,0,1, 1,1,1,0,1,0,0,0, 0,0,0,1,0,0,1,1, 0,1,0,1,0,1,0,0,
```

```
   0,0,0,0,1,1,1,1, 0,0,0,0,1,0,1,0, 1,0,1,1,0,1,0,0, 0,0,0,0,0,1,0,1]

expansionR :: [Int] -> [a] -> [a]
expansionR [] _      = []
expansionR (e:xs) ys = (ys!!(e-1)) : expansionR xs ys

expansion :: [Int] -> [a] -> [a]
expansion xs ys = foldr (\x y-> (ys!!(x-1)):y) [] xs

expansionGen :: [a] -> Gen [Int]
expansionGen xs =
  do k <- shuffle [1..length xs]
     return (k++k)

prop_Expansion :: String -> Property
prop_Expansion xs =
  forAll (expansionGen xs)
         (\q -> expansionR q xs == expansion q xs)

pc_1 :: [Int]
pc_1 = [57,49,41,33,25,17,9,1,58,50,42,34,26,18,10,2,59,51,43,35,27,19,11,3,60,
        52,44,36,63,55,47,39,31,23,15,7,62,54,46,38,30,22,14,6,61,53,45,37,29,
        21,13,5,28,20,12,4]

mitades :: [a] -> ([a],[a])
mitades xs = splitAt (length xs `div` 2) xs

desplazaIzq :: Int -> [Int] -> [Int]
desplazaIzq n xs = drop n xs ++ take n xs

desplazamientos :: [Int]
desplazamientos = [1,2,4,6,8,10,12,14,15,17,19,21,23,25,27,28]

pc_2 :: [Int]
pc_2 = [14,17,11,24,1,5,3,28,15,6,21,10,23,19,12,4,26,8,16,7,27,20,13,2,41,52,
        31,37,47,55,30,40,51,45,33,48,44,49,39,56,34,53,46,42,50,36,29,32]

crea16Claves :: [Int] -> [[Int]]
crea16Claves ks =
  [expansion pc_2 k | k <- ks'']
```

```
  where (xs,ys) = mitades ks'
        ks'     = expansion pc_1 ks
        ks''    = [ desplazaIzq n xs ++ desplazaIzq n ys
                  | n <- desplazamientos]

ip :: [Int]
ip = [58,50,42,34,26,18,10,2,60,52,44,36,28,20,12,4,62,54,46,38,30,22,14,6,
      64,56,48,40,32,24,16,8,57,49,41,33,25,17,9,1,59,51,43,35,27,19,11,3,
      61,53,45,37,29,21,13,5,63,55,47,39,31,23,15,7]

ex :: [Int]
ex = [32,1,2,3,4,5,4,5,6,7,8,9,8,9,10,11,12,13,12,13,14,15,16,17,16,17,18,19,
      20,21,20,21,22,23,24,25,24,25,26,27,28,29,28,29,30,31,32,1]

s1, s2, s3, s4, s5, s6, s7, s8 :: Matrix Int
s1 = fromLists [[14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7],
                [0,15,7,4,14,2,13,1,10,6,12,11,9,5,3,8],
                [4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0],
                [15,12,8,2,4,9,1,7,5,11,3,14,10,0,6,13]]

s2 = fromLists [[15,1,8,14,6,11,3,4,9,7,2,13,12,0,5,10],
                [3,13,4,7,15,2,8,14,12,0,1,10,6,9,11,5],
                [0,14,7,11,10,4,13,1,5,8,12,6,9,3,2,15],
                [13,8,10,1,3,15,4,2,11,6,7,12,0,5,14,9]]

s3 = fromLists [[10,0,9,14,6,3,15,5,1,13,12,7,11,4,2,8],
                [13,7,0,9,3,4,6,10,2,8,5,14,12,11,15,1],
                [13,6,4,9,8,15,3,0,11,1,2,12,5,10,14,7],
                [1,10,13,0,6,9,8,7,4,15,14,3,11,5,2,12]]

s4 = fromLists [[7,13,14,3,0,6,9,10,1,2,8,5,11,12,4,15],
                [13,8,11,5,6,15,0,3,4,7,2,12,1,10,14,9],
                [10,6,9,0,12,11,7,13,15,1,3,14,5,2,8,4],
                [3,15,0,6,10,1,13,8,9,4,5,11,12,7,2,14]]

s5 = fromLists [[2,12,4,1,7,10,11,6,8,5,3,15,13,0,14,9],
                [14,11,2,12,4,7,13,1,5,0,15,10,3,9,8,6],
                [4,2,1,11,10,13,7,8,15,9,12,5,6,3,0,14],
                [11,8,12,7,1,14,2,13,6,15,0,9,10,4,5,3]]
```

```
s6 = fromLists [[12,1,10,15,9,2,6,8,0,13,3,4,14,7,5,11],
                [10,15,4,2,7,12,9,5,6,1,13,14,0,11,3,8],
                [9,14,15,5,2,8,12,3,7,0,4,10,1,13,11,6],
                [4,3,2,12,9,5,15,10,11,14,1,7,6,0,8,13]]

s7 = fromLists [[4,11,2,14,15,0,8,13,3,12,9,7,5,10,6,1],
                [13,0,11,7,4,9,1,10,14,3,5,12,2,15,8,6],
                [1,4,11,13,12,3,7,14,10,15,6,8,0,5,9,2],
                [6,11,13,8,1,4,10,7,9,5,0,15,14,2,3,12]]

s8 = fromLists [[13,2,8,4,6,15,11,1,10,9,3,14,5,0,12,7],
                [1,15,13,8,10,3,7,4,12,5,6,11,0,14,9,2],
                [7,11,4,1,9,12,14,2,0,6,10,13,15,3,5,8],
                [2,1,14,7,4,10,8,13,15,12,9,0,3,5,6,11]]

divideEn8 :: [Int] -> [[Int]]
divideEn8 [] = []
divideEn8 xs = take 6 xs : divideEn8 (drop 6 xs)

sCaja :: ([Int],Matrix Int) -> [Int]
sCaja ([a,b,c,d,e,f],s) = int2bin4 (s!(i,j))
  where i = 1 + bin2int4 [a,f]
        j = 1 + bin2int4 [b,c,d,e]
sCaja _ = []

p :: [Int]
p = [16,7,20,21,29,12,28,17,1,15,23,26,5,18,31,10,2,8,24,14,32,27,3,9,19,13,
     30,6,22,11,4,25]

funcionF :: [Int] -> [Int] -> [Int]
funcionF k r =
  expansion p (concat [ sCaja a
                      | a <-zip (divideEn8 (xor (expansion ex r) k))
                                [s1,s2,s3,s4,s5,s6,s7,s8]])

rondaFeistel :: [Int] -> [Int] -> [Int]
rondaFeistel k m = r ++ xor l (funcionF k r)
  where (l,r) = mitades m

rondasFeistel :: [Int] -> [Int] -> [Int]
```

```
rondasFeistel ks = aux (crea16Claves ks)
  where aux [] m      = snd (mitades m) ++ fst (mitades m)
        aux (k:ks') m = aux ks' (rondaFeistel k m)

invIP :: [Int]
invIP =
  [40,8,48,16,56,24,64,32,39,7,47,15,55,23,63,31,38,6,46,14,54,22,62,30,
   37,5,45,13,53,21,61,29,36,4,44,12,52,20,60,28,35,3,43,11,51,19,59,27,
   34,2,42,10,50,18,58,26,33,1,41,9,49,17,57,25]

encriptaDES' :: [Int] -> [Int] -> [Int]
encriptaDES' ks m
  | length m == 64 =
      expansion invIP (rondasFeistel ks (expansion ip m))
  | otherwise = m

encriptaDES :: String -> String -> String
encriptaDES ks m =
  descodifica (aux (codifica ks) (codifica m) [])
  where aux _ [] xs = xs
        aux k ms xs =
          aux k (drop 64 ms) (xs ++ encriptaDES' k (take 64 ms))

rondasFeistelInv :: [Int] -> [Int] -> [Int]
rondasFeistelInv ks = aux (reverse(crea16Claves ks))
  where aux [] m      = snd (mitades m) ++ fst (mitades m)
        aux (k:ks') m = aux ks' (rondaFeistel k m)

desencriptaDES' :: [Int] -> [Int] -> [Int]
desencriptaDES' ks m
  |length m == 64 =
     expansion invIP (rondasFeistelInv ks (expansion ip m))
  | otherwise = m

desencriptaDES :: String -> String -> String
desencriptaDES ks m =
  descodifica (aux (codifica ks) (codifica m) [])
  where aux _ [] xs = xs
        aux k ms xs =
          aux k (drop 64 ms) (xs ++ desencriptaDES' k (take 64 ms))
```

```
claveYmensajeDES :: Gen (String,String)
claveYmensajeDES =
  do n <- choose (1,6)
     k <- vector 8
     m <- vector (8*n)
     return (k,m)

prop_CorreccionDES :: Property
prop_CorreccionDES =
  forAll claveYmensajeDES
         (\(k,m) -> m == desencriptaDES k (encriptaDES k m))
```

A.12. Asociacion4

```
strToInteger1 :: String -> Integer
strToInteger1 cs =
  foldl (\x y -> toInteger y + 1000 * toInteger x) 0 (map ord cs)

strToInteger :: String -> Integer
strToInteger = read . concatMap aux
  where aux c | c < '\n' = "00" ++ cs
              | c < 'd'  = '0' : cs
              | otherwise = cs
          where cs = show (ord c)

prop_strToInteger :: String -> Property
prop_strToInteger m =
  m /= [] ==> strToInteger1 m == strToInteger m

integerToStr1 :: Integer -> String
integerToStr1 0 = []
integerToStr1 n = integerToStr1 (div n 1000)
                  ++ [chr (fromIntegral (rem n 1000))]

integerToStr :: Integer -> String
integerToStr m | x == 0    = aux s
               | x == 1    = chr (read (take 1 s)) : aux (drop 1 s)
               | otherwise = chr (read (take 2 s)) : aux (drop 2 s)
```

```
  where x     = rem (length (show m)) 3
        s     = show m
        aux n = map (chr . read) (chunksOf 3 n)

prop_integerToStr :: Integer -> Property
prop_integerToStr n =
  n>0 ==>  integerToStr1 n == integerToStr n

prop_strToInteger_integerToStr :: String -> Property
prop_strToInteger_integerToStr m =
  m /= [] && 0 /= (ord . head) m ==>
    m == (integerToStr . strToInteger) m

prop_integerToStr_strToInteger :: Integer -> Property
prop_integerToStr_strToInteger n =
  n > 0 ==> n == (strToInteger . integerToStr) n

expQuickMod :: Integer -> Integer -> Integer -> Integer
expQuickMod x 1 n = x `mod` n
expQuickMod x e n
  | even e    = expQuickMod (x*x `mod` n) (div e 2) n
  | otherwise = (x * expQuickMod x (e-1) n) `mod` n

prop_Exp :: Integer -> Integer -> Integer -> Property
prop_Exp x e n =
  e > 0 && n > 0 ==>
    expQuickMod x e n == x^e `mod` n

coprimos :: Integral a => a -> a -> Bool
coprimos a b = gcd a b == 1
```

A.13. RSA

```
import Asociacion1 (invMod)
import Asociacion4 (strToInteger, integerToStr, coprimos, expQuickMod)
import Codec.Crypto.RSA.Exceptions
import  "crypto-api" Crypto.Random
import Crypto.Random.DRBG
import Data.Char (ord)
```

```
import System.IO.Unsafe
import Test.QuickCheck

ejemploE :: Integer -> Gen Integer
ejemploE phiN =
  suchThat (choose (1,phiN)) (coprimos phiN)

ejemploE' :: Integer -> Integer
ejemploE' phiN =
  unsafePerformIO (generate (ejemploE phiN))

clavePubYPrivRSA :: Int -> IO ((Integer, Integer),(Integer, Integer))
clavePubYPrivRSA m = do
  g1 :: GenAutoReseed HashDRBG HashDRBG <- newGenIO
  let (p,g2) = largeRandomPrime g1 m
      (q,_)  = largeRandomPrime g2 m
      n = p*q
      phiN = (p-1)*(q-1)
      e = ejemploE' phiN
      d = invMod e phiN
  return ((n,e),(n,d))

clavePubYPrivRSA' :: Int -> ((Integer, Integer), (Integer, Integer))
clavePubYPrivRSA' m =
  unsafePerformIO (clavePubYPrivRSA m)

encriptaRSA ::  String -> (Integer,Integer) -> String
encriptaRSA m (n,e) = integerToStr (expQuickMod (strToInteger m) e n)

desencriptaRSA :: String -> (Integer,Integer) -> String
desencriptaRSA = encriptaRSA

prop_CorrecRSA :: Int -> String -> Property
prop_CorrecRSA x m =
  m /= [] && x>0 && x<30 && (ord (head m) > 0) &&
  (length m < div ((length . show) a) 3) ==>
     m == desencriptaRSA (encriptaRSA m (a,b)) (c,d)
  where ((a,b),(c,d)) = clavePubYPrivRSA' x
```

A.14. ElGamal

```
import Asociacion4 (strToInteger, integerToStr, expQuickMod)
import Data.Char (ord)
import Data.Text (chunksOf,pack,unpack)
import Codec.Crypto.RSA.Exceptions
import  "crypto-api" Crypto.Random
import Crypto.Random.DRBG
import Factory.Math.PrimeFactorisation
import Factory.Math.Implementations.PrimeFactorisation
import Test.QuickCheck
import System.IO.Unsafe

baseG :: Integer -> Integer -> Integer
baseG p q = unsafePerformIO (generate (baseG' p q))
  where baseG' a b = suchThat (choose (1,a-1))
                              (\x -> 1 /= expQuickMod x (div (a-1) b) a)

generaEntorno :: Int -> (Integer, Integer, Integer)
generaEntorno m = unsafePerformIO (generaEntorno' m)
  where generaEntorno' n = do
          g1 :: GenAutoReseed HashDRBG HashDRBG <- newGenIO
          let (p,_) = largeRandomPrime g1 n
              (q,_) = last (primeFactors TrialDivision (p-1))
              g     = baseG p q
          return (p,q,g)

clavePubyPrivElGamal :: (Integer, t, Integer)
                     -> ((Integer, Integer, Integer),(Integer,Integer))
clavePubyPrivElGamal x = unsafePerformIO (clavePubyPrivElGamal' x)
  where clavePubyPrivElGamal' (p,_,g) = do
            let n = unsafePerformIO (generate (choose (1,p-1)))
            return ((p,g,expQuickMod g n p),(p,n))

encriptaElGamal :: String
                   -> (Integer, Integer, Integer) -> [(String, String)]
encriptaElGamal m (a,b,c) = foldr
                   (\z y-> encriptaElGamal' (unpack z) (a,b,c) : y) []
                           (chunksOf (div ((length . show) a) 3) (pack m))
 where encriptaElGamal' x y = unsafePerformIO  (encriptaElGamal'' x y)
```

```
        where encriptaElGamal'' mensaje (p,g,e) = do
               let h = unsafePerformIO (generate (choose (1,p)))
               return (integerToStr (expQuickMod g h p),
                       integerToStr (m' * expQuickMod e h p `mod` p))
               where m' = strToInteger mensaje

desencriptaElGamal :: [(String, String)] -> (Integer, Integer) -> [Char]
desencriptaElGamal xs (a,b) = foldr
             (\m y -> (desencriptaElGamal' m (a,b) ++ y)) [] xs
 where desencriptaElGamal' (mensaje1,mensaje2) (p,n) =
          integerToStr (expQuickMod m1 (p-1-n) p * m2 `mod` p)
             where (m1,m2) = (strToInteger mensaje1, strToInteger mensaje2)

prop_CorrecElGamal1 :: String -> Property
prop_CorrecElGamal1 m =
  m /= [] && all (\ x -> ord x > 0) m ==>
    m == desencriptaElGamal (encriptaElGamal m clavPub) clavPriv
      where (clavPub,clavPriv) =
             clavePubyPrivElGamal (909209565115226705262277 :: Integer,
                                   17530839358873691 :: Integer,
                                   803233093626684455375681 :: Integer)

prop_CorrecElGamal2 :: Int -> String -> Property
prop_CorrecElGamal2 n m =
  m /= [] && all (\ x -> ord x > 0) m  ==>
    m == desencriptaElGamal (encriptaElGamal m clavPub) clavPriv
  where (clavPub,clavPriv) = clavePubyPrivElGamal (generaEntorno n)
```

A.15. Rabin

```
import Asociacion4 (strToInteger, integerToStr)
import Data.Char (ord)
import Codec.Crypto.RSA.Exceptions
import  "crypto-api" Crypto.Random
import Crypto.Random.DRBG
import Math.NumberTheory.Moduli (sqrtModFList)
import System.IO.Unsafe
import System.Random
import Test.QuickCheck
```

```
choose' :: (Integral a, Random a) => a -> a
choose' n = unsafePerformIO (aux n)
  where aux m = getStdRandom (randomR (1,div m 2))

prim3Mod4 :: CryptoRandomGen t => Int -> (Integer, t) -> Integer
prim3Mod4 m (p,g) | p `mod` 4 == 3 = p
                  | otherwise = prim3Mod4 m x
  where x = largeRandomPrime g m

clavesPubyPrivRabin :: Int ->
             ((Integer,Integer),(Integer,Integer,Integer,Integer))
clavesPubyPrivRabin n = unsafePerformIO (aux n)
  where aux m = do
          g1 :: GenAutoReseed HashDRBG HashDRBG <- newGenIO
          g2 :: GenAutoReseed HashDRBG HashDRBG <- newGenIO
          let (p0,g3) = largeRandomPrime g1 m
              p       = prim3Mod4 m (p0,g3)
              (q0,g4) = largeRandomPrime g2 m
              q       = prim3Mod4 m (q0,g4)
              n'      = p*q
              e       = fromIntegral (2 * choose' n)
          return ((n',e),(p,q,n',e))

encriptaRabin1 :: (Integer, t) -> String -> String
encriptaRabin1 (n,_) ms = integerToStr ((m*m) `mod` n)
  where m = strToInteger ms

desencriptaRabin1 :: (Integer, Integer, Integer, t) -> String -> [String]
desencriptaRabin1 (p,q,n,_) ms =
  foldr (\x y -> integerToStr (x `mod` n) : y) [] raices2
  where m       = strToInteger ms
        raices2 = sqrtModFList m [(p,1),(q,1)]

encriptaRabin :: (Integer, Integer) -> String -> String
encriptaRabin (n,e) ms = integerToStr ((m*(m+e)) `mod` n)
  where m = strToInteger ms

desencriptaRabin :: (Integer, Integer, Integer, Integer) -> String -> [String]
desencriptaRabin (p,q,n,e) ms =
```

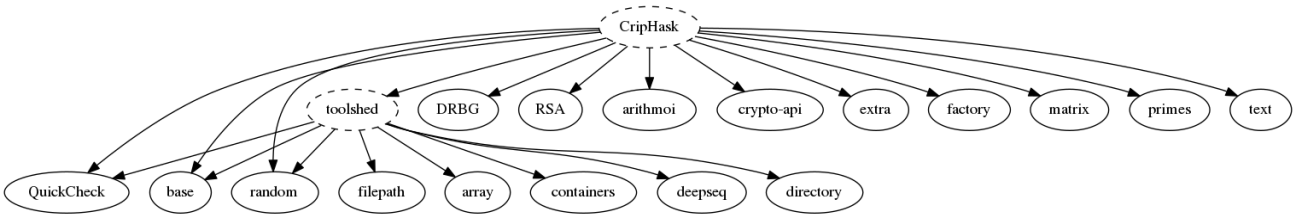
```
  foldr (\x y -> integerToStr ((x-e_2) `mod` n) : y) [] raices2
  where e2_4    = div (e * e) 4
        e_2     = div e 2
        m       = strToInteger ms
        raices2 = sqrtModFList ((m + e2_4) `mod` n) [(p,1),(q,1)]

prop_CorrecRabin :: Int -> String -> Property
prop_CorrecRabin x m =
  m /= [] && 0 < x && x < 30 && (ord (head m) > 0) &&
  (length m < div ((length . show) a) 3) ==>
    elem m (desencriptaRabin (c,d,e,f) (encriptaRabin (a,b) m))
  where ((a,b),(c,d,e,f)) = clavesPubyPrivRabin x
```

Apéndice B

Dependencias

En el siguiente grafo se representan las dependencias entre los módulos desarrollados en el trabajo

En el siguiente grafo se representan las dependencias entre las librerías utilizadas en el trabajo



Bibliografía

Textos

1. R.S. Boyer y J S. Moore: *Proof checking the RSA public key encryption algorithm*. The American Mathematical Monthly Vol. 91, No. 3 (Mar., 1984), pp. 181–189.

2. K. Claessen y M.H. Pałka: *Splittable pseudorandom number generators using cryptographic hashing*. En: Proceedings of the 2013 ACM SIGPLAN symposium on Haskell, Haskell '13, ACM, 2013, pp. 47–58.

3. J. Duan, J. Hurd, G. Li, S. Owens, K. Slind y J. Zhang: *Functional correctness proofs of encryption algorithms*. En G. Sutcliffe y A. Voronkov, editorws, Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2005), volume 3835 of Lecture Notes in Artificial Intelligence. Springer-Verlag, December 2005.

4. L. Erkök et als.: *Programming Cryptol*. Galois, 2015.

5. S. Goldwasser y M. Bellare: *Lecture notes on cryptography*. Massachusetts Institute of Technology (MIT), 2008.

6. D. Gray: *Implementing public-key cryptography in Haskell*. School of Computer Applications Dublin City University, Tech. Rep., November, 12 2001.

7. C. Lindenberg, K. Wirt y J. Buchmann: *Formal proof for the correctness of RSA–PSS*. Cryptology ePrint Archive, Report 2006/011, http://eprint.iacr.org

8. A. Lochbihler: *Probabilistic functions and cryptographic oracles in higher order logic*. En "European Symposium on Programming Languages and Systems", pp. 503–531, (2016), Springer.

9. M.J. Lucena López: *Criptografía y seguridad en computadores*. Universidad de Jaen, 2010.

10. G. Márton: *Public-key cryptography in functional programming context*. Acta Univ. Sapientiae, Informatica, 2, 1 (2010) 99–112.

11. A.J. Menezes, P.C. van Oorschot y S.A. Vanstone: *Handbook of applied cryptography*. CRC press, 1996.

12. J.J. Ortiz Muñoz: *Criptografía y matemáticas*. Revista Suma, N. 61 (Junio 2009) pp. 17-26.

13. R. Rivest, A. Shamir y L. Adleman: *A method for obtaining digital signatures and public-key cryptosystems*. Communications of the ACM CACM. Volume 21 Issue 2, Feb. 1978, pp. 120–126.

14. S. Singh: *The code book: The science of secrecy from ancient Egypt to quantum cryptography*. Anchor , 2000.

15. S. Singh: *Los códigos secretos: el arte y la ciencia de la criptografía, desde el antiguo Egipto a la era Internet*. Editorial Debate, 2000.

16. E.W. Smith y D.L. Dill: *Automatic formal verification of block cipher implementations*. En: Proc. of the 2008 International Conference on Formal Methods in Computer-Aided Design (FMCAD 2008), pp. 1–7 (2008)

17. Wikibooks: *Cryptography*. https://en.wikibooks.org/wiki/Cryptography

18. Maria Mañeru: *El enigma de los códigos secretos*. Editorial Libsa, S.A., 2014.

Cursos

1. M. Bellare: *Introduction to modern cryptography*. http://cseweb.ucsd.edu/~mihir/cse207/classnotes.html

2. R. Pass y A. Shelat *A course in cryptography* http://www.cs.cornell.edu/courses/cs4830/2010fa/lecnotes.pdf

3. S.A. Weis *Theory and practice of cryptography mini-course* http://saweis.net/crypto.html

4. D. Wu *Introduction to cryptography* http://www.cse.chalmers.se/edu/course/TDA351/lectures/CS255LectureNotes.pdf

5. Departamento de Álgebra de la Universidad de Sevilla: *Apuntes de la asignatura de Teoría de Códigos y Criptografía*

Páginas Web

1. Wikipedia.
   En https://es.wikipedia.org
2. Exercitium (Ejercicios de programación funcional con Haskell)
   En http://www.glc.us.es/~jalonso/exercitium/
3. Detexify.
   En http://detexify.kirelabs.org/classify.html
4. Hayoo! - Haskell API Search.
   En http://hayoo.fh-wedel.de/
5. Hoogle:a Haskell API search engine
   En https://www.haskell.org/hoogle/
6. Cifrado Hill
   En http://materias.fi.uba.ar/6669/docs/Hill.pdf
7. Criptografía
   En http://www.textoscientificos.com/criptografia
8. CryptoHill
   En http://cryptohill.blogspot.com.es/
9. Cifrar Online
   En https://cifraronline.com
10. Miniejercicios con LaTeX
    En http://minisconlatex.blogspot.com.es

Índice alfabético